The work was conducted under contract DASW01-94-C-0054, Task T-S5-1266, for the Defense Information Systems Agency. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

PREFACE

This document was prepared by the Institute for Defense Analyses (IDA) under the task order, Object-Oriented Technology Implementation in the Department of Defense (DoD), in response to a task objective to develop strategies for the implementation of object-oriented technology (OOT) within specific information technology areas within the DoD. This document is one of a set of four reports on OOT implementation. The other reports, focusing on other areas of OOT, are IDA Paper P-3142, *Object-Oriented Development Process for Department of Defense Information Systems*; IDA Paper P-3144, *Legacy System Wrapping for Department of Defense Information System Modernization*; and IDA Paper P-3145, *Software Reengineering Using Object-Oriented Technology.* All of this work was sponsored by the Defense Information Systems Agency (DISA).

The following IDA research staff members were reviewers of this document: Dr. Edward A. Feustel, Dr. Richard J. Ivanetich, Dr. Reginald N. Meeson, Dr. Judy Popelas, Mr. Clyde G. Roby, and Mr. Glen R. White.

A particular word of thanks is expressed to Mr. David Diskin of DISA for his involvement and support of this effort. The editorial assistance of Ms. Sylvia W. Reynolds and Ms. Katydean Price, along with secretarial support by Ms. Paula L. Giffey and Ms. Leslie J. Norris, is gratefully acknowledged.

Table of Contents

List of Figures

EXECUTIVE SUMMARY

Purpose

This report describes alternative techniques for implementing object-oriented (OO) software using the Ada programming language. It depicts the problems and solutions of OO programming using the 1983 Ada programming language standard (Ada 83) and presents an overview of the basics of OO programming using the 1995 standard on Ada (Ada 95). Its targeted audience is primarily software engineers, though it may be of interest to managers interested in the technical details of OO programming in Ada.

Background

Object-oriented technology (OOT) has demonstrated special capabilities to help meet Department of Defense (DoD) needs for software reuse, streamlined system development, systems interoperability, and reduction of maintenance and modification costs. In addition, OOT is maturing in areas such as OO software engineering methodology, OO programming languages, computer-aided software engineering tools, OO database management systems, and OO standards. Thus, OOT is in a good position to aid the massive migration and reengineering of DoD software systems from many outmoded systems to fewer, modernized, interoperable, less costly systems. However, the transition from traditional software technologies to OOT is not trivial: a substantial learning period is required before the benefits of OOT can be realized. This report aims to facilitate that learning process by describing specific techniques for OO programming using the Ada programming language.

Ada was developed by the DoD to provide a standard programming language for implementing software for embedded systems, replacing the proliferation of numerous proprietary languages that had previously dominated that application area. Its original design goals included readability for ease of maintenance and reuse, strong typing for improved reliability, encapsulation and library support for “programming in the large,” and data abstraction to facilitate portability and maintenance. While the standard for Ada approved in 1983 by the American National Standards Institute realized these particular goals, it falls

short of providing direct support for several of the essential features of OO programming, including class-attribute extension and polymorphism. Although the new version of Ada, Ada 95, has been developed to rectify these deficiencies (among other enhancements), its full capabilities are not yet available in validated compilers. Thus, for an interim period, OO programming for systems requiring the use of validated Ada compilers will need to program in these missing OO features using Ada 83. However, as this report was being written, progress has continued on Ada 95 compilers and validation suites, so that new use of Ada 83 may be phased out in the relatively near future.

Scope

This report restricts its attention to the Ada programming language, and it addresses programming strategies for both Ada 83 and Ada 95. It describes alternative implementation techniques for the essential features of OO programs: classes, objects, inheritance, polymorphism, and associations. It does not consider OO implementation in programming languages other than in Ada and focuses on Ada 83, which poses the most challenges to the OO programmer because of its incomplete support for OO features. However, as mentioned previously, Ada 83 is fast becoming obsolete due to the advent of Ada 95 and forthcoming validation of full Ada 95 compilers.

While inheritance implementation techniques are described here, they only cover single inheritance in which a class has at most one superclass since Ada does not provide support for multiple inheritance. The report does not address more specialized issues in OO implementation such as distributed processing, OO databases, and OO frameworks.

Programming Strategies

The *object-based* Ada 83 language, unlike the *object-oriented* Ada 95, provides for several principal strategies for implementing the OO programming features of inheritance and polymorphism. Two of the implementation strategies are described in detail, and the basic OO features of the fully object-oriented Ada 95 revision are explained. One Ada 83 OO programming strategy is based on using variant records to support the addition of new attributes to subclasses of an Ada class. This strategy is recommended for less experienced Ada programming teams since it is type safe and less complex. However, the combined degree of modularity and encapsulation supported by this strategy is limited. The other, more complex strategy, which we call a class-wide strategy, achieves a higher combined degree of modularity and encapsulation through the use of unchecked type conversion. This

strategy is recommended for experienced Ada programming teams because of its superior modularity and encapsulation which aids both maintainability and any transition to Ada 95.

Which strategy is best will depend on the specifics of the project. Projects with inexperienced Ada programming teams may do better to adopt the simpler strategy to reduce the risks of programming errors, while more experienced Ada programmers may confidently implement a more complex strategy.

1. INTRODUCTION

1.1 PURPOSE AND SCOPE

This report describes alternative strategies for implementing object-oriented (OO) systems using the Ada programming language. Since validated Ada compilers are only available for the 1983 standard of the programming language (Ada 83), current Department of Defense (DoD) Ada projects may have to continue to use it despite its lack of some essential OO features. Thus, any such projects using an OO approach will have to select a strategy for implementing these missing OO features. This report explains two of the most promising such strategies to facilitate that selection process and subsequent OO programming in Ada. In addition, the fundamentals of OO programming appearing in the new, fully object-oriented Ada standard (Ada 95) are explained.

1.2 BACKGROUND

When building an OO system, it is most natural to use an OO programming language. However, complete validated compilers for the latest version of the Ada programming language, Ada 95, are not yet available. At this writing, the available validated compilers for the Ada programming language only accommodate the Ada 83 standard, which is not a fully OO language but is only "object based." As a result, the transition from an OO design model to its implementation in Ada 83 is not as straightforward as it would be in fully OO languages such as Smalltalk, C++, or Ada 95. Thus, when Ada 83 is used for an OO information system, some of the OO features missing from it will have to be programmed in order to implement an OO system design.

We digress briefly to examine the differences between object-oriented and merely object-based languages before returning to discuss their importance with respect to OO programming in the two versions of the Ada language. A convenient definition of a "object-oriented programming language" developed by Cardelli and Wegner [CAWE85] is as follows:

> ". . . a language is object-oriented if and only if it satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state
- Objects have an associated type [class]
- Types [classes] may inherit attributes from supertypes [superclasses]."

Object-based languages fall short of satisfying this definition only in their lack of sufficient support for inheritance. More specifically, while Ada 83 does support inheritance of attributes and operations through derived types, it does *not* support the extension of the *attributes* of a base type with additional attributes in any of its derived types, although it *does* support the extension and revision of *operations* in derived types. The absence of support for data (or attribute) extension in Ada 83 sharply limits its support for reuse because extensions to existing classes are not easily accomplished through inheritance if new attributes need to be added.

Another limitation of Ada 83 which keeps it from qualifying as fully object oriented is the absence of support for dynamic polymorphism. An operation in a programming language is polymorphic if it can be applied to different classes, using different procedures in different cases [RUMB91, p. 25]. Full polymorphism obviates any need for case statements to laboriously check each alternative type of argument of an operation to determine the appropriate procedure based on its type. Different classes may have different procedures defined for the same operation name, which are called automatically when the operation is called with arguments of that type of class. Such polymorphism considerably simplifies OO programming and makes it especially amenable to extension since additional data types and associated procedures may be added to a program without changing existing calls to the operation. Ada 83 does not directly support dynamic run-time polymorphism in which the data type of an argument to an operation can be determined while the program is running. Despite the absence of direct support, polymorphism can be programmed into Ada 83 at different levels of fidelity to a polymorphic ideal. Ada 95 overcomes the limitations of Ada 83 with its introduction of a special data type, the *class-wide type*, which can take any class from within a given class hierarchy as a value.

The addition of class-wide types and true single inheritance to Ada in Ada 95 make it a true OO language. Compilers for Ada 95, however, are still undergoing development. While the language standard has been set, the process of developing validation suites and validating compilers is still in progress as of this writing. The whole process of developing commercial compilers, validation suites, and validating them for the full Ada 95 language may not be completed for another couple of years. In the interim, some DoD systems may

be developed or migrated using the existing programming language standard, Ada 83. Thus, if we are to consider using object-oriented technology (OOT) in DoD software development over the year or so, we must consider how to adapt Ada 83 to the needs of OO programming. In this report we will review briefly some of the possible mappings between an OO design model and the Ada language, in both its Ada 83 and Ada 95 versions, to show how this transition may be achieved.

1.3 APPROACH

The programming techniques described here are based upon known approaches that have appeared in the Ada programming literature. This document serves primarily to select some of the most promising techniques and to group them together to cover all the basic aspects of OO programming in Ada. Code fragments that are provided have all been tested for compilation in the context of complete examples.

1.4 ORGANIZATION OF DOCUMENT

Chapter 1 has described the limitations of Ada 83 support for OO programming. In Chapter 2, criteria are described for evaluating alternative strategies for OO programming in Ada 83. This is followed in Chapter 3 by a presentation of the most common strategy, using Ada's variant records to implement class hierarchies. The limitations of this strategy are identified as motivation for a more elaborate approach using unchecked type conversion to better emulate the OO approach taken in Ada 95. Chapter 4 provides an overview of OO programming in Ada 95. A strategy for OO programming in Ada 83 that closely emulates the approach of Ada 95 is described in Chapter 5. Chapter 6 discusses alternative strategies (applicable to both Ada 83 and Ada 95) for the implementation of associations between classes or between specific objects. Appendix A contains complete example code for a simplified employee tax calculation program to fully illustrate this technique of "class-wide" programming in Ada 83. While this technique provides for better encapsulation, modularity, and ease of transition to Ada 95, it also carries the costs of greater complexity and risk. Programming conventions are identified that will mitigate this risk if faithfully followed. Finally, lists of references, glossary, acronyms, and an index are provided.

2. CRITERIA FOR OO PROGRAMMING IN ADA 83

As with any engineering problem, trade-offs must be evaluated to find an optimum solution to OO programming in Ada 83. The following general criteria apply when evaluating strategies for implementing OO programming using Ada 83:

- Format simplicity - ease of translation of an OO design to Ada 83.
- OO faithfulness - preservation of intended OO semantics in Ada 83.
- Memory conservation - no memory leaks as objects de-referenced.
- Modification simplicity - ease of maintenance and extension of code.
- Transition simplicity - ease of transition from Ada 83 to Ada 95.
- Safety - OO programming techniques not prone to errors or failure.
- Conformance - OO programming strategy works with accepted software engineering practices.

Simplicity is always a criterion for any engineering product since a simpler product is ordinarily less costly to produce and often more reliable than a more complex one. The first item, format simplicity, identifies a preference for one type of simplicity, the simplicity of translation of an OO design into Ada 83. Naturally, this translation will not be as simple for Ada 83 as for fully object-oriented languages, but there is considerable variation in simplicity among alternative strategies for OO implementation in Ada 83. How well a strategy satisfies this criterion will be the principal factor determining the time it takes to implement an OO design.

OO faithfulness is the property of preserving the general semantic properties of the OO paradigm in the translation from design to implementation. Ideally, the semantics should be identical. However, depending on the design method and the language, subtle differences can lead to divergences from this ideal. This is especially true when the programmer attempts to simulate an OO feature such as polymorphism. Every variable (object) must maintain its identity and uniqueness. In OO programming, an object may be referenced from several places in a program, through data structures that implement associa-

tions, or through lists of objects of various types. Any change to an object must be visible globally. This is usually implemented in the form of some unique identifier such as an index into an array or an access type (pointer) in Ada 83. In such cases, all operations on the object should be implemented as operations on the identifier and all references to the object should use the identifier. Strategies for implementing classes in Ada differ on the ease with which this semantics of uniqueness is supported in objects: some integrate it into the object definition while others require greater care in programming to avoid creating duplicates of the same object.

Memory conservation requires the proper construction and destruction of objects to ensure that memory leaks are not introduced when objects are de-referenced (i.e., when all references to an object are eliminated). Several OO languages, such as Ada 95, automatically support memory management because of its technical difficulties and the practically universal need to control memory leakage in OO implemented systems.

Modification simplicity is another simplicity criterion where the interest is in the ease of modification to code for purposes of maintenance, extension, or reuse in other systems or programs. There are few systems that never change. To realize one of the key benefits of OO technology, the implementation must also be amenable to change. Key factors affecting the ease of modification are the degree of modularity afforded by the implementation strategy, especially when augmenting the attributes and operations of classes in their subclasses.

Transition simplicity is concerned with the ease of transition from Ada 83 to Ada 95. Can the implementation move easily to this new version of the Ada language? Since new Ada 95 compilers are forthcoming, an implementation strategy should be prepared to exploit the new language features and, if possible, easily convert to the new constructs. For example, the Ada 95 language revision introduces child packages which can be simulated in Ada 83 using corresponding naming conventions for its packages and types, as discussed in Chapter 5 on the strategy of simulating Ada 95's class-wide types.

The safety of an OO programming technique depends on its liability for contributing to programming errors in a particular context. For example, unchecked conversion between datatypes in Ada is a less safe programming practice than one that conforms to Ada's strict typing rules. The results of such unchecked conversions are compiler dependent and can be unpredictable when moving to a new compiler, although they are unproblematic in many circumstances in most existing Ada environments. Strict enforcement of

programming conventions governing such practices (presented throughout Chapter 5) can reduce such risks, as discussed in the conclusions of Chapter 5.

Satisfying the conformance criterion occurs when the OO programming strategy works well with other accepted practices of software engineering. In particular, when programming in Ada, it can aid comprehension and communication within a programming team if common style guidelines for Ada programming are observed.

Encapsulation, inheritance, and polymorphism are the three major features of object-oriented design and programming. The following sections examine these features in more detail and explain the relevant criteria for successful implementation of them.

2.1 ENCAPSULATION

Ada provides strong encapsulation semantics through its packaging mechanism. The Ada package is a basic Ada program unit that supports selective encapsulation of the data and procedures it contains. Packages are coded in two parts, a specification and a body, in the following general format:

```
package <package_name>  is
       [<declaration>]*
[private
       [<declaration>]*]
end  <package_name>;
```

package specification

```
package body  <package_name>  is
      [<declaration>]*
[begin <sequence_of_statements>
      [exception <exception_handler>*]]
end  <package_name>;
```

package body

The specification may have a private part which is not visible outside of the package. This is often used to encapsulate the specification of internal structure of class attributes since the private part is not visible outside of the package (except for its child packages in Ada 95, as explained in Section 4.3). The public part of the specification declares all the data types and operations that are visible outside the package. This is where the access and update functions for attributes are specified, along with specifications for any other operations that are accessible outside of the package. The package body contains the code

implementing class operations, including attribute access and update operations. A package body is also encapsulated since its code is not visible beyond its bounds and can be accessed only through its package specifications. The optional sequence of executable statements in the body is rarely used in practice.

Packages provide the structure within which a class is implemented. A single data type is declared in the public (non-private) part of the package specification to serve as the data type for all of the objects in a class. The structure of this class data type, however, is typically specified in the private part of the specification to preserve its encapsulation. Attributes are ordinarily specified in a data structure accessed by (or identical with) the class data type. Class operations are declared in the specification as procedures or functions taking variables of the class data type as arguments (among others, as required). The code implementing the operations of a class is encapsulated in the body of the corresponding package. This much about implementing classes in Ada 83 is uncontroversial. However, a variety of different strategies have been developed for implementing the details of such class representations, including variations in the form of the class data type, level of encapsulation, in inheritance techniques, and in approaches to polymorphism. The preferred strategies for handling these details are controversial.

As far encapsulation is concerned, each approach should be evaluated on the basis of how well it supports the following:

- Strong encapsulation - conventional or language enforced?
- Variety of encapsulation - is encapsulation available in different strengths?
- Full modularity - does encapsulation affect modularity?

Encapsulation may be strictly enforced by the language by hiding class information in the body and private parts of a class's package. However, in some approaches, such strict encapsulation blocks direct access to subclass attributes from the packages defining a subclass. Thus strict encapsulation can result in violations of modularity when it requires placing the access and update functions for attributes unique to a subclass in its base superclass. Full modularity would require keeping all (sub)class specific operations with the specific (sub)class. This specific problem is explained in Chapter 3 in the course of presenting the common variant record approach to OO class implementation. More generally, the strict encapsulation and strong typing of Ada 83 require a trade-off between encapsulating the implementation details of a class and allowing other modules (packages) to extend the class. An alternative to strict encapsulation is *encapsulation by programming*

convention; although the language allows access to attribute structures, the programming conventions for the system disallow it. Provided the attributes are always accessed and updated via dedicated functions, they are effectively encapsulated in that system. This alternative allows greater modularity within the context of the Ada 83 variant record strategy for implementing classes.

Another issue in encapsulation is whether a strategy supports multiple levels of encapsulation. Some OO languages allow designated classes to have access to their internal attribute structures or operations, such as the class "friends" in C++ or "children" packages in Ada 95. Implementing such variable visibility in Ada 83 is difficult, though it could be achieved via different levels of access conventions.

Although a package specification defines the class interface, encapsulation can be used for purposes other than implementing OO classes. Consequently, not all Ada packages need to define a class. For example, an attribute may take on values that require non-trivial operations, such as a person's name. Identifying the last, first, middle, title, etc., and maintaining these distinct parts may be considered to constitute enough complexity to warrant separate encapsulation in a separate package. This is commonly referred to as an *abstraction* and will, in practice, look similar, if not identical, to class encapsulation. Allowance of such abstractions is another dimension of the variety of encapsulation supported by an OO implementation strategy. There are, however, no requirements involving uniqueness and reference semantics as found in classes. In fact, the behavior should be documented in the package specification to avoid confusion. For example, if a person's name is also the attribute of some roster, will the name change on the roster when the title field of the name is updated?

2.2 INHERITANCE

Ada 83 provides a restricted form of static inheritance which can work in some cases to implement inheritance found in an OO design. The language construct is called a derived type and takes the form:

```
type <Child Type> is new <Parent Type>;
```

All of the operations that work on the parent type are inherited and new operations can be added or inherited operations can be overridden. However, the child type cannot extend its data structure to hold additional attribute values. This limitation has been a significant difficulty for programmers trying to implement inheritance in Ada 83. Ada 95 over-

comes this limitation by allowing attribute extensions for types declared as **tagged**, as further discussed in Chapter 4. One of the principal challenges in implementing OO designs in Ada 83 is programming in an effective means of adding additional attributes to a child class.

2.3 POLYMORPHISM

With polymorphism, the particular operation invoked by a call on an operation can depend on the particular kind of object being operated on. A call to a payroll calculation operation, for example, may invoke different operations, depending on the type of employee, e.g., full-time, part-time, or consulting. (Annotated code for an extended example of this type appears in Appendix A. Further description of the code is provided in a companion report on software wrapping [IDA95c].) Polymorphism may be supported to varying degrees by different approaches to OO programming. Different features of polymorphism, all of which are expected of any fully polymorphic system, can be distinguished as follows:

- Static polymorphism - class and operation determined at compile time.
- Dynamic polymorphism - class/operation determined at run time.
- Modularity maintained - operations defined locally in classes.
- Encapsulation maintained - class attribute structures defined as private.
- Inheritance respected - most specific operation in hierarchy selected.
- Automatic dispatching - assigned to the appropriate operation.

A type of static polymorphism is directly supported by operation overloading in all versions of Ada, which allows the same operation name to be used for different versions of the operation and is distinguished at compile time by the data types of its arguments. For example, an Ada 83 program might make separate calls to different procedures for different types of employees, all named “Tax_calc”. Although a convenient feature, this is not the commonly accepted definition of polymorphism. The more common form of dynamic polymorphism can be simulated in Ada 83 and will be available in Ada 95.

Ada 83 can support dynamic run-time polymorphism through the use of variant records and explicit type conversion among derived types. This common approach can be found throughout introductory Ada texts. We describe the essentials of this approach in the next chapter. However, the limitations of this approach are also well documented and typically compromise either modularity or encapsulation. This is why both modularity and

encapsulation are included as separate criteria for effective implementation of polymorphism. It is difficult to implement dynamic polymorphism in Ada 83 while maintaining the full encapsulation and modularity characteristic of the best OO programming.

A higher degree of modularity and full encapsulation of classes is possible in Ada 83, using an approach that more closely follows that taken by Ada 95. However, this strategy of emulating the Ada 95 class structure achieves its advantages at some costs. It is considerably complicated by the widespread use of unchecked conversions between subclasses and superclasses, a procedure that is inherently risky and compiler dependent. Such risks can be greatly mitigated, however, by strict conformance with programming practices which restrict unchecked conversion to localized sections of class definitions. This strategy for implementing polymorphism is described in Chapter 5, after the Ada 95 approach is first described in Chapter 4.

Implementations of polymorphism may also vary in the extent of their support for inheritance. Inheritance requires that when an operation is chosen for a specific class, it is always the most specific operation for that class from all the operations of that name that might be defined for the ancestors of a class. For example, if an employee were classified as a part-time administrative staff member, the class might inherit its tax calculation from the class of all part-time employees rather than from the generic class of employee. This ability to choose the most specific class when selecting the operations for a class-wide type at run time could be implemented for the general case or may require hand adjustments to the dispatching code whenever a more specific operation is added. Typically, OO implementation in Ada 83 requires these sorts of hand adjustments to detailed case statements whenever a polymorphic operation is added since the implementation of automatic inheritance is awkward and adds to the run time overhead. While automatic selection of most specific inherited operations is supported by Ada 83 through its built-in inheritance mechanism, that selection only applies to subclass-specific datatypes, not to the class-wide programming used in polymorphism. The strategy outlined in the next chapter incorporates the common approach requiring such hand adjustments for polymorphic dispatching.

3. VARIANT RECORD CLASSES IN ADA 83

In this chapter, we review the most common strategy for implementing OO programming in Ada 83 which uses variant records to implement class hierarchies. We begin with a simplified template in Section 3.1 for representing a class in Ada 83 to identify the basic structure of a single class package. A simple example of implementing an employee class is presented in Section 3.2 to illustrate the use of such a template. Section 3.3 on inheritance shows how a class template can be elaborated with a variant record to represent a hierarchy of classes. An elaboration of the employee class example using a variant record for a hierarchy of employee classes is given next in Section 3.4, followed by a description of the implementation of polymorphism under this strategy in Section 3.5. Finally, Section 3.6 presents a set of conclusions regarding the variant record strategy for OO programming in Ada 83.

3.1 PRELIMINARY CLASS TEMPLATE

3.1.1 Template

Programming conventions on the organization of class packages, such as the placement of different kinds of operations, can ease coding and comprehension during implementation and subsequent phases. Many such conventions can be conveniently represented in the form of a programming template (or schema) providing a format for specifying classes, as in the following simple class template for Ada 83:

```
package <package_name>  is
     type <class> is [limited] private;
     -------------- Other data type declarations:----------------------
     [<declaration>]*
     -------------- Attribute access operations:-----------------------
     function <attribute-1> (Self: in <class>)
                            return <attribute-1_type>;
           ....
     function <attribute-n> (Self: in <class>)
                            return <attribute-n_type>;
```

```
-------------- Attribute update operations:----------------------
procedure Change  (Self         : in out <class>;
                   <attribute-1> : in <attribute-1_type>);
      ....
procedure Change  (Self: in out <class>;
                   <attribute-n> : in <attribute-n_type>);
------------- Class construction & destruction operations:--------
procedure Construct
          ([<attribute-i> : in <attribute-i_type>]
           [;<parameter_specification>]*)
          return <class>;
procedure Destruct (Self: in out <class>);
-------------- Other class operations: --------------------------
procedure <operation-1> (Self : in out <class>
                         [;<parameter_specification>]*);
....
function <operation-k>  (Self : in <class>
                         [;<parameter_specification>]*)
          return <result-type-k>);
--------- Private section of Ada package specification: ----------
private
      [<declaration>]*   ------ Any private data definitions
--------- Class attributes defined in private section: -----------
      type <class> is record
          <attribute-1> : <attribute-1_internal_type>;
          <attribute-2> : <attribute-2_internal_type>;
          ....
          <attribute-n> : <attribute-n_internal_type>;
      end record;
end  <package_name>;
```

This is just one of the simplest of a wide variety of possible conventions for representing classes in Ada 83. It is used here just to introduce the basic elements of implementing classes in Ada 83. Since it lacks support for augmenting the attributes of subclasses and for polymorphism, more elaborate conventions that support these capabilities will be introduced in the sections on inheritance (Section 3.3) and polymorphism (Section 3.5).

3.1.2 Class Declaration

After specifying the package name, this template declares the distinguishing class data type as private with the private type declaration:

```
type <class> is private;
```

The full definition of the class data type then occurs below the declaration in the private section of the package. This ensures encapsulation of the internal representation of attribute data that is stored in the instances of this data type. An appropriate class name is substituted for the placeholder "<class>" in all places it occurs for any specific instance of this class specification template, in accord with naming conventions selected for a system. Alternative forms for class data type definitions are discussed in Section 3.1.6.

Since data types in Ada are relative to the programming unit within which they are declared, the full name for a class in Ada is composed of the package name and the class data type, e.g., <package_name>.<class>. These composite names for Ada classes require establishing a translation convention to map class names from the design models of analysis and design to the implemented classes in Ada. One possible approach maps the design model class names directly to the package names and identifies every class data type with the term "Class". An employee class, for example, could be implemented in a package named "Employee" with the class type named "Class", yielding the full class name "Employee.Class". This is unproblematic, provided only one class is defined in a package. Exceptions might be made for multiple classes in a package, or a convention could be enforced restricting each package to contain a single top-level class. Alternatively, the class type in Ada could use the original class name, and package names might be related or the same, as in "Employee.Employee", though that would be redundant. Whatever approach is taken, it is helpful to have a system-wide convention to ease comprehension and coding.

When the class being defined is a subclass, it is customary in Ada to define it as a derived type, using a derived type data definition such as the following:

```
type <sub_class> is new <class>;
```

This allows the subclass type to inherit all the attributes and operations of the parent class using Ada's built-in support for inheritance. Some elaboration of this approach is required if subclasses are to be extended with additional attributes in Ada 83, as is discussed later in Section 3.3 on implementing inheritance.

3.1.3 Other Data Type Declarations

Following the declaration of the class data type in this template are any other public data type declarations that might be needed. This is where data definitions of the public values of attributes may be specified if they are not built-in types or defined in other included packages. While the internal form of a class attribute is ordinarily private, some external

(public) form is required if attribute values are to be accessible outside the private parts of a class package. The external data types of attributes are used in the public functions that access or update them, as illustrated by the references of the form <attribute-n-type> in the attribute access and change operations of the template. In many cases, it may prove convenient to allow the external and internal data types for attributes to be identical.

3.1.4 Operation Specifications

Data type declarations are followed by all the public class operation specifications. The class template presented at the beginning of this section organizes operations into the categories of access, update, construct and destruct, and other. The basic operations of access, update, construct, and destruct do not need all be present in every class since they may be inherited from superclasses or defined in subclasses. Abstract classes, in particular, should not have construct or destruct operations since they are not intended to have their own instances but to provide a basis for subclasses which have instances. Access and update operations should be defined for any attributes whose values are to be publicly accessible.

All operations specified in the object model of a class should be specified within the section of "other class operations". If for any reason the design model operations are reassigned to a subclass or superclass during implementation, then the design model should be updated to coordinate it with the implementation.

Attribute access operations for returning the values of attributes are specified first in this template, followed by attribute update (or change) operations. Access operations are all specified as functions of the following form:

```
function <attribute-i>  (Self : in <class>)
          return <attribute-i_type>;
```

Functions are well suited for this purpose since they are the most natural operation for simply returning values, though procedures might also be used. The template suggests using a name for each attribute access function that describes the attribute being accessed, e.g., "salary" for the salary attribute of an employee. These functions all take an instance of the class as an operand, and return the value of the attribute in its external form of (<attribute-i_type>). Ordinarily, each attribute will have an access function, though it might be omitted if an attribute is only used internally by other operations of its object.

Update or change operations for attributes are all specified in this template as procedures of the following form:

```
procedure Change  (Self          : in out <class>;
                   <attribute-n> : in     <attribute-n_type>);
```

Procedures are used for update operations since a return value, as provided by a function, is not ordinarily required. In this class template, a single procedure name “Change” is overloaded to name each of the separate update procedures for every attribute. Any particular call to these change procedures is disambiguated by the type of its second argument (<attribute-n_type>) which identifies the appropriate attribute. A simple alternative to this approach could include the attribute names in the update operation names (e.g., “Change_salary”), thereby allowing multiple attribute values to have the same datatype. The first argument to these update procedures is always an instance of the class with a mode of “in out” allowing the input object to be changed upon output. The value of the second argument in a particular call to a Change procedure is used to replace any existing value of the attribute. While this class template specifies one change procedure for each attribute, alternative conventions might allow changes to multiple attributes with a single change procedure or even omit change procedures for certain static attributes.

Constructor and destructor operations are provided to construct and destroy instances of a class. Under the presented class template, constructors may be procedures of the following form:

```
procedure Construct
      ([<attribute-i> : in <attribute-i_type>]
       [;<parameter_specification>]*)
       Self : in <class>);
```

Functions might be used instead, although they would be problematic if the class data type were specified to be “limited” since Ada blocks assignment for such data types. A constructor may take attribute values and other parameters as input in order to assign certain attribute values or establish associations. Alternatively, a constructor might not have parameters, leaving attribute assignments to update functions. The basic activity of a constructor operation is to declare a variable to be an instance of the class type. While such declarations could take the place of calling constructor operations for creating simple objects, extensibility and maintainability are simplified by the use of explicit constructor operations. Constructors also provide a convenient format for initializing the values of object attributes and for establishing associations with other objects.

Destructors are conveniently implemented as procedures since they do not return values. They need only take a single parameter of the type of the class, as follows:

```
procedure Destruct (Self: in out <class>);
```

A destructor may be responsible for deallocating memory for the object and proper removal of any associations it may have with other objects. While Ada 83 relies upon the implementor to call object destructors when needed, Ada 95 supports special controlled data types which a programmer may use to ensure that an appropriate destructor is called when leaving the scope of an object's declaration.

Other class operations are specified following the basic operations of access, update, construct, and destruct. These "other" operations are ordinarily the only ones specified in the OO analysis and design models since suitable basic operations for access, update, construction, and destruction may be assumed for any class. The "other" operations may be formulated as either procedures or functions, depending on whether they are designed to return values and how they are used. In either case, it is a convenient convention to require the first argument or operand of the operation to be an object in the class. Procedures may thus take the following form:

```
procedure <operation-1>  (Self : in out <class>
                          [;<parameter_specification>]*);
```

Such procedures or functions can take any number of other objects or data values as parameters, depending on their designed functionality.

3.1.5 Private Specifications

The private section of the package specification may begin with any definitions of datatypes that need not be visible outside of the package. Such definitions are ordinarily only required for the internal data types used in representing class attributes if they are not defined elsewhere.

The key feature in the private section of this class specification template is the definition of the class data type. Ordinarily, such data types are composite types since they are intended to contain all the attributes of each class, and it is rare to design a class with only one attribute. The following class template specifies the class data type as a record consisting of a set of *n* attributes in the format:

```
type <class> is record
    <attribute-1> : <attribute-1_internal_type>;
    <attribute-2> : <attribute-2_internal_type>;
    ....
    <attribute-n> : <attribute-n_internal_type>;
end record;
```

Each attribute in this record is provided with a name (<attribute-i>), and its internal data type is defined (<attribute-i_internal_type>). It often proves convenient to allow the internal data type of an attribute to be identical to the external datatype of its access values. Even under this condition, it remains helpful to keep the class record definition in the private part of the specification. It is good OO programming practice to always access attributes through their access functions in order to minimize the disruption of the code when changes to the internal representation are required. Such encapsulation ensures that only the access and update operations need change when the internal representation changes. However, it is possible to place the record definition in the public part of the package and retain effective encapsulation by enforcing the programming convention that attributes are only to be accessed or updated by their access or update operations.

Here it is assumed that each class attribute will correspond to one element in the record of the class data type. This is one of the simplest conventions for storing attribute values. Alternative conventions for representing attributes might allow either combining multiple attributes into a single record or data value or deriving an attribute value from one or more of the record values. However, it is generally good practice to distinguish stored attributes from derived ones to avoid confusing interactions amongst them. A direct update to a derived value, for example, could have unpredictable consequences for the values of the attributes from which it is derived. Therefore, a derived value can be better represented by a calculation operation on its class rather than by a distinct attribute.

3.1.6 Alternative Class Data Structures

A useful variation on defining class data types as records is to define them as pointers to record structures. Pointers afford a certain ease in manipulating class instances (objects) in many contexts such as in object lists and object associations. Their use helps ensure that the uniqueness semantics of individual objects are preserved throughout operations involving them. In Ada, pointers are implemented by a special data type called an *access data type*, and are specified separately from the objects to which they point, as in the following template for the private part of a class data type definition:

```
type Structure is record
      <attribute-1> : <attribute-1_internal_type>;
      ....
      <attribute-n> : <attribute-n_internal_type>;
end record;
type <class> is access Structure;
```

A variant on this, which can achieve the same effects, retains the class data definition as a record structure but defines pointers to such structures as an additional data type, as in the following:

```
package <package_name>  is
     ------------ Class type declared private: ------------------------
     type <class> is private;
     type <class>_pointer is access <class>;
               ....
     ---------- Private section of Ada package specification: ---------
     private
                 ....
           type <class> is record
              <attribute-1> : <attribute-1_internal_type>;
              ....
              <attribute-n> is <attribute-n_internal_type>;
           end record;
     end  <package_name>;
```

Either of these approaches may be used to support multiple access to the same object, though the former approach may better enforce the exclusive use of pointers since the "Structure" of the pointers may be defined only in the private section, hence not accessible outside of the package. This can be advantageous in typical programming contexts in which objects should be referenced only by pointers in order to ensure that they are not inadvertently copied.

3.2 PRELIMINARY EXAMPLE OF A CLASS

A specific example of an Ada 83 package specification fitting one variant of the templates presented in the previous section can be coded for a simplified version of an employee class (Employee.Class) as follows:

```
with ADAR_Comp;
package Employee is
  --------------- Employee.Class type declaration: ------------------
  type Class is private;
  -------------- Public Attribute Type Declarations: ----------------
  type Name  is new String (1..25);
  type Social_Security_Number is new String(1..11);
  type Money is new ADAR_Comp.Decimal (Precision => 9, Scale => 2);
```

```
    -------------- Attribute access operations:----------------------
    function Emp_Name   (Self : in Class) return Name;
    function SS_Number  (Self : in Class) return Social_Security_Number;
    function Emp_Salary (Self : in Class) return Money;
    -------------- Attribute update operations:----------------------
    procedure Change (Self      : in Class;  Emp_Name : in Name);
    procedure Change (Self      : in Class;
                      SS_Number : in Social_Security_Number);
    procedure Change (Self      : in Class;  Salary   : in Money);
    ----------- Class construction & destruction operations:----------
    function Construct (Emp_Name    : in Employee.Name;
                        SS_Number   : in Social_Security.Number)
      return Class;
    -------------- Other class operations: --------------------------
    function Net_Pay       (Self : in Class) return Money;
    function Send_Check_To (Self : in Class) return String;
    -------------- Encapsulated attribute formats: ------------------
  private
    type Structure is
             record
                   Emp_Name     : Name;
                   SS_Number    : Social_Security_Number;
                   Emp_Salary   : Money;
             end record;
    type Class is access Structure;
  end Employee;
```

This example uses the ADAR_Comp library package to provide decimal arithmetic definitions for use in defining and manipulating a data type (Money) suitable for representing an employee's salary. This attribute and the other employee.class attributes are defined publicly and used for both internal and external representations. In addition to basic access, change, and construction operations attributes, this package defines a *Net_Pay* operation to calculate an employee's net pay, and a *Send_Check_To* operation to determine a location for sending a paycheck. Various elaborations of this basic definition to better support inheritance and polymorphism are discussed in Sections 3.4 and 3.5, respectively.

3.3 INHERITANCE

As noted previously, Ada 83 supports the inheritance of attributes and operations by the subclasses (derived types) of a class (type). It also allows the addition of new operations to subclasses. It falls short of full support for single inheritance in its failure to support the addition of new attributes to subclasses. Fortunately, there are several possible workarounds for this shortcoming that will allow the extension of attributes in subclasses or at least the appearance of such extensions. Otherwise, OO programming in Ada 83 would be impractical since extensions of class attributes by subclasses are a common characteristic of OO analysis and design models.

3.3.1 Simple Inheritance

Perhaps the simplest approach to supporting additional attributes in subclasses for Ada 83 programs is one that supports such appearances while actually retaining the same data structure throughout a class hierarchy. One such approach includes all attributes that are used by a base class or any of its derived classes in the data record of the base class, but this approach only supports operations for subclass specific attributes within those subclasses. Access and update operations for a subclass-specific attribute, for example, might only be specified and defined in the subclass. This approach ensures that subclass attributes are effectively invisible in superclasses where they do not belong, although it will be wasteful of storage whenever subclasses do extend the attributes of the base class. It also violates the modularity expected of OO systems since the base class definition has to be augmented whenever a subclass is added with new attributes.

Encapsulation will also be compromised in this simplest approach if the access and update operations of subclasses are defined within them. Since the attribute structure is fully defined in the base (or root) ancestor class, it will not be visible to subclasses defined in different packages if it is in the private section. Thus, if the access and update operations for subclass-specific attributes are to be defined in that subclass, the attribute structure needs to be left visible in the parent structure, preventing its full encapsulation. Encapsulation of a sort may still be achieved by enforcing the programming convention that all attributes are to be accessed only by their access and update functions (except within those functions themselves).

Encapsulation enforced by the private section of the package structure in Ada may be retained in this sort of approach if the attribute access and update functions are all kept in the base class where the attributes are defined. However, this tactic results in less modu-

larity since subclass-specific access and update operations are no longer defined within the subclass package. Consequently, there is a trade-off in such approaches between modularity and encapsulation: You cannot have both full modularity and language-enforced encapsulation using this type of approach.

3.3.2 Variant-Record Inheritance

The storage inefficiencies of our simplest approach can be remedied by defining the base class datatype using a variant record with a record discriminant to identify the specific subclass. An Ada variant record allows a single record type to have different components, depending upon the value of a discriminant which is supplied as an argument to the record type. We can apply this to discriminate different class records based upon the class type by defining a discriminant that ranges over the different types of classes in a given hierarchy. While this is the common approach to OO programming in Ada 83, it continues to suffer from the trade-off of modularity and encapsulation described above for the simplest approach. Either the class structure must be made public or the attribute access and update operations must be restricted to the base class.

A template for a base class (<base_package>.<class>) supporting variant record class hierarchies can be defined as follows:

```
package <base_package>  is
     ---------- Class type declarations ------------------------------
     type Tag is (Base, Sub1, Sub2,..., Subs);
     type Structure (Tagged : Tag) is private;
     type <class> is access Structure;
     .....
-------- Private section of Ada package specification: ---------------
private
     [<declaration>]*         ------ Any private declarations]
     --------- Class attributes defined in private section: ----------
     type Structure(Tagged :Tag) is record
           <attribute-1> : <attribute-1_internal_type>;
           ....
           <attribute-n> : <attribute-n_internal_type>;
           case Tagged is
               when Base => null;
               when Sub1 =>
                  <attribute-1-1> : <attribute-1-1_internal_type>;
```

```
                ....
                <attribute-1-k1> : <attribute-1-k1_internal_type>;
            when...
            when Subs =>
                <attribute-1-1> : <attribute-1-1_internal_type>;
                ....
                <attribute-1-ks> : <attribute-1-ks_internal_type>;
        end case;
    end record;
end  <base_package>;
```

The type *Tag* is declared as an enumeration type with values (Base, Sub_1, Sub_2,..., Sub_s) naming the different subclasses of this base class including one (Base) to identify the base class itself. A data structure (*Structure*) for the class, using this the discriminant *Tagged* of this type (*Tag*), is declared to be defined in the private section. This structure is kept private in this template to illustrate the coding for maximal encapsulation. An example will be given shortly illustrating the alternative of keeping the structure visible but encapsulated by its access and update operations, allowing greater modularity.

In the private section of the class specification, the structure of the class is defined using a variant record. The variant record definition begins with attributes that are common to all objects in the base class, followed by a case statement that identifies any additional attributes that are required by subclasses.

Subclasses derived from this base class can then be defined using the following template:

```
with <base_package>[, <package>]*;
package <sub_package>  is
      ---------- Class type declared tagged private  -------------
      type <sub_class> is new <package>.<super_class>;
      .....
--------- Private section of Ada package specification: ----------
[private
      [<declaration>;]* ] ------ Any private definitions
end  <sub_package>;
```

When the subclass is a child (immediate descendant) of the base class, then its definition need only *with* the base package (<base_package>), and the child's type is specified as a new type of the base class type, i.e.,

```
type <sub_class> is new <base_package>.Class;
```

When the subclass is not a child of the base class, the packages of all its intermediate superclasses must also be *with*'d to ensure access to their type definitions, although all operations can be inherited by just *with*'ing the immediate superclass (i.e., the parent).

To select the proper set of attributes for a particular subclass from the variant record structure of such a class hierarchy, the constructors for subclasses should instantiate the discriminant value to the corresponding *Tag* value. This discriminant cannot be set in the type definition or we would be unable to have more than one level of subclasses because once the discriminant is set, it cannot be changed. An example of how a construct function for the i^{th} subclass Sub_i might be defined is as follows:

```
function Construct (<attribute-1> : <attribute-1_type>;...
                    <attribute-1-j1> : <attribute-1-j1_type>)
        return Class is
        New_object :
            Class(new <base_package>.Structure
                          (<base_package>.Subi));
        begin
            Change(New_object, <attribute-1>);...
            Change(New_object, <attribute-1-j1>);
            return New_object;
        end;
```

The parameters to the Construct function are typically used to set attribute values, as indicated, though they might also be used for setting associations or for other purposes. Here the specified attributes are set using the attributes' update procedures (possibly inherited) to fully maintain their encapsulation.

As with our simplest approach to attribute extension, any subclass-specific operations (apart from access and update operations) may be defined within the subclass package. Individual objects constructed to be of a particular class will now have a record structure that includes all and only those attributes appropriate to it. Thus, the creation of unused attribute components characteristic of the simplest approach is avoided. This template does suffer some redundancy in the specification of attributes in the variant record of the base class due to the flat format of its case statement. New attributes of subclasses will have to be repeated in the cases of any of their derived classes. This redundancy could be avoided by a more complex template that creates a nesting of case statements reflecting the class hierarchy, as described in [BAKR91]. However the variant record is defined, modularity remains violated since any new attributes of subclasses must be defined in the corre-

sponding base class.

3.4 EXAMPLE: ADA 83 PACKAGE SPECIFICATION

A specific example of an Ada 83 package specification fitting the above variant record template for a simplified version of an employee class (Employee.Class) is given as follows:

```
with ADAR_Comp;
package Employee is
 --------------- Employee.Class type declarations ----------------
 type Tag is (Base, Consulting, Salaried, Hourly);
 type Structure (Tagged : Tag) is private;
 type Class is access Structure;
 -------------- Public Attribute Type Declarations ---------------
 type Name  is new String (1..25);
 type Social_Security_Number is new String(1..11);
 type Money is new ADAR_Comp.Decimal (Precision => 9, Scale => 2);
 type Deduction is range 0..12;
 type Mailing_Address is access String;
 -------------- Attribute access operations:-----------------------
 function Emp_Tag (Self : in Class) return Tag;
 function Emp_Name   (Self : in Class) return Name;
 function SS_Number  (Self : in Class) return Social_Security_Number;
 function Emp_Salary (Self : in Class) return Money;
 function Emp_Deduction  (Self : in Class) return Deduction;
 function Mail_Address (Self : in Class) return  String;
 -------------- Attribute update operations:-----------------------
 procedure Change (Self      : in Class;  Emp_Name : in Name);
 procedure Change (Self      : in Class;
                   SS_Number : in Social_Security_Number);
 procedure Change (Self      : in Class;  Salary   : in Money);
 procedure Change (Self      : in Class;  Deduction: in Deduction);
 procedure Change (Self      : in Class;  Mail_Address : in String);
---- Class construction & destruction operations: (in subclasses)---
-------------- Other class operations: ----------------------------
function Net_Pay       (Self : in Class) return Money;
```

```
function Send_Check_To (Self : in Class) return String;
--------------- Encapsulated attribute formats ---------------------
private
   type Structure(Tagged :Tag) is
         record
         Emp_Name     : Name;
         SS_Number    : Social_Security_Number;
         Emp_Salary   : Money;
         case Tagged is
             when Base => null;
             when Consulting =>
                   MA     : Mailing_Address;
             when Salaried | Hourly =>
                   Tax    : Money;
                   D      : Deduction;
         end case;
     end record;
end Employee;
```

This example uses the ADAR_Comp library package to provide decimal arithmetic definitions for use in defining and manipulating a data type (Money) suitable for representing an employee's salary. The datatypes for this attribute and the other employee.class attributes are defined publicly and are used for both internal and external representations. Attribute access and update operations are defined here for all subclasses because the class structure is defined in the private section. In addition to basic access, change, and construction operations attributes, this package declares a *Net_Pay* operation to calculate an employee's net pay and a *Send_Check_To* operation to print an employees check.

3.5 POLYMORPHISM

Polymorphism is simulated within the variant record strategy by simply adhering to the following additional conventions:

- Writing operation calls in terms of the base class without specifying a discriminant value, and
- Implementing a case statement to dispatch a polymorphic operation to the appropriate subclass operation, depending on the discriminant value at run time.

Then, a base class variable can function as a class-wide type, and the discriminants appropriate to any of its base classes may assigned to it when the function call is applied to a

specific case.

3.5.1 Example Base Class

An example of polymorphism and dispatching under this OO programming strategy can be adapted from the examples of a system for processing alerts (alarms) in a ground mission control center as found in [BARN93a, BARN93b]. A package for the class of alerts could be specified as the following:

```
with Calendar,...;
package Alert is
  ---------- Class type declared private: ------------------------
  type Tag is (Base, Low, Medium, High);
  type Structure (Tagged : Tag) is private;
  type Class is access Structure;
  -------------- Public Attribute Type Declarations: --------------
  type Device is (Teletype, Console, Big_Screen);
  -------------- Attribute access operations: ----------------------
  function Arrival_time (Self : in Class) return Calendar.Time;
  function Message_text (Self : in Class) return Text;
  -------------- Attribute update operations:----------------------

   procedure Change (Self: in Class;  Arrival_time : in Calendar.Time);
   procedure Change (Self: in Class; Message_text : in Text);
  ---- Class construction & destruction operations: in subclasses ---
  -------------- Other class operations: --------------------------
  procedure Handle(Self  : in Class);
  procedure Base_Handle(Self : in Class);
  procedure Display(Self : in Class; On: in Device);
  procedure Log(Self     : in Class);
  procedure Set_Alarm(Self : in Class);
  --------------- Attribute structure: ----------------------------
     type Structure(Tagged :Tag) is record
           Time_Of_Arrival: Calendar.Time;
           Message: Text;
           case Tagged is
               when Base => null;
               when Low  => null;
               when Medium | High =>
```

```
                Action_Officer: Person;
              when High  =>
                Ring_Alarm_At: Calendar.Time
          end case;
       end record;
    type Class is access Structure;
 end  Alert;
```

Three different kinds of alerts (*Low*, *Medium*, and *High*) are specified in the variant record case statement giving the alert class attribute structures. Class operations, such as *Handle*, can then be implemented in the package body to dispatch to appropriate subclass as follows:

```
with Low_Alert, Medium_Alert, High_Alert;
package body Alert is
.........
procedure Handle(Self : in Class) is
begin
     case Self.Tagged is
           when Low => null;
           when Medium =>
               Medium_Alert.Handle(Medium_Alert.Class(Self));
           when High =>
               High_Alert.Handle(High_Alert.Class(Self));
     end case;
end Handle;
end Alert;
```

The package body must include the subclass packages (*Low_Alert, Medium_Alert, High_Alert*) using a "with" statement in order to have access to their operations for dispatching. Notice that the non-specific type *Alert.Class* is narrowed to the specific subclass, depending on its discriminant tag, before passing it to the appropriate subclass operation.

3.5.2 Subclass Specializations of Operations

Operations specific to particular subclasses are implemented in the individual subclass operations. The subclass implementations here will reflect the subclass relations wherein each higher-level alert is a subclass of the prior-level alert. Thus, the bodies of the subclasses may implement the *Handle* operation as follows:

```
with Alert;
package body Low_Alert is
```

```
     procedure Handle(Self : in Class) is
     begin
           Change(Self, Arrival_time => Calendar.Clock);
           Log(Self);
           Display(Self, Teletype);
     end Handle;
end Low_Alert;

with Low_Alert;
package body Medium_Alert is
.........
     procedure Handle(Self : in Class) is
     begin
           Low_Alert.Handle(Low_Alert.Class(Self));
           Change(Self, Action_Officer => Assign_Volunteer);
           Display(Self, Console);
     end Handle;
end Medium_Alert;

with Medium_Alert;
package body High_Alert is
.........
     procedure Handle(Self : in Class) is
     begin
           Medium_Alert.Handle(Medium_Alert.Class(Self));
           Display(Self, Big_Screen);
           Set_Alarm(Self);
     end Handle;
end High_Alert;
```

Each successive subclass here can make use of the alarm handling procedure of its immediate superclass because these procedures are supersets of each other. Notice, however, that the class data type has to be converted to one matching the superclass whenever one of its overridden operations is called. This nesting of subclass operation definitions illustrates the versatility of the general implementation strategy, although polymorphic subclass operations may have entirely distinct implementations in other cases.

3.5.3 Subclass Specifications

Because the attribute definition was made public in the base class (*Alert.Class*), the

subclass specifications for different alerts can include all of their distinctive access and update operations as well as the *Handle* operation that is defined differently for each.

```
with Alert,...;
package Low_Alert is
  type Class is new Alert.Class;
  ------------- Class construction & destruction operations:----------
  function Construct (Message : Text) return Class;
  -------------- Other class operations: ----------------------------
  procedure Handle(Self  : in Class);
end  Low_Alert;

with Alert, Low_Alert;
package Medium_Alert is
  type Class is new Low_Alert.Class;
  ------------ Attribute access operations:--------------------------
  function Action_Officer (Self : in Class) return Person;
  -------------- Attribute update operations:-------------------------
  procedure Change (Self : in Class; Action_Officer : in Person);
  ------------- Class construction & destruction operations:----------
  function Construct (Message : Text) return Class;
  -------------- Other class operations: ----------------------------
  procedure Handle(Self  : in Class);
end  Medium_Alert;

with Alert, Low_Alert, Medium_Alert;
package High_Alert is
  type Class is new Medium_Alert.Class;
  ------------ Attribute access operations:--------------------------
  function Ring_Alarm_At (Self : in Class) return Calendar.Time;
  -------------- Attribute update operations:-------------------------
  procedure Change(Self : in Class; Ring_Alarm_At : in Calendar.Time);
  ------------- Class construction & destruction operations:----------
  function Construct (Message : Text; Ring_Alarm_At : Calendar.Time)
           return Class;
  -------------- Other class operations: ----------------------------
  procedure Handle(Self  : in Class);
end  High_Alert;
```

The *Low_Alert* package need not include any attribute operations since all its attributes are common to all alerts and defined in the base class *Alert*. Each subclass must define its own construction operation in order to create objects in that class with the appropriate discriminant tag and corresponding record structure.

3.6 CONCLUSIONS

Our examples illustrate some of the flexibility and modularity achievable with this variant record strategy for OO programming in Ada 83.

> *The variant record strategy for OO programming in Ada 83 can support the basic OO features of encapsulation, inheritance, and polymorphism. However, the combined degree of modularity and encapsulation achievable with this strategy remains limited by its reliance on variant records. Either a high-level of modularity or a high-degree of encapsulation can be achieved, but both cannot be achieved simultaneously.*

If the variant records are encapsulated in the private section of a class package, then modularity suffers since subclass-specific attribute access and update procedures cannot be localized to the subclasses. If the maximal modularity is to be achieved, then encapsulation must be downgraded to being enforced by convention, as it is in the given examples. An alternative strategy for OO programming in Ada 83, based on the Ada 95 approach, is possible that avoids this particular trade-off, achieving a high level of encapsulation and modularity. This alternative will be described shortly after the basic OO features of Ada 95 are introduced in Chapter 4

4. ADA 95

Ada 95 is the recently revised standard for Ada which has been designed to better accommodate OO programming among its other enhancements. The OO enhancements in Ada 95 provide several new features that affect the way classes are best represented in the language.

4.1 TAGGED AND CLASS-WIDE TYPES

The principal change in Ada 95 that affects the implementation format for classes is the introduction of new datatype capabilities in the *tagged* type and its associated class-wide type. A tagged type is specified using the new reserved word "tagged" in type definitions, as follows:

```
type <class> is tagged record
    <component_list>
end record;
```

Defining a class in this manner identifies it as a type that can be extended with new data components in any of its subclasses.

A subclass can be defined as a derived type with additional data components as follows:

```
type <sub_class> is new <package>.<class> with record
    <new_component_list>
end record;
```

Alternatively, a subclass can be defined with no new components by specifying the record to be null, i.e.,

```
type <sub_class> is new <package>.<class> with null record;
```

All subclasses of a tagged type are also considered tagged, and a single class-wide type, of the form <class>'Class, may be used to refer to any of them. While a class-wide type can refer to any of its subclasses (derived types), it is an unconstrained type that must be initialized to a specific subclass when used by itself. However, an access type pointing to a class-wide type does not need to initialize the class-wide type. Hence, such access types

are used in Ada 95 to provide dynamic polymorphism in which the specific subclass of the class wide type need not be determined until runtime.

4.2 BASIC CLASS TEMPLATE

Tagged types are recommended for implementing class data types within Ada 95 because of the support they provide for augmenting attributes and for polymorphism. Modifying the latest variant-record class template to use tagged types in Ada 95 requires marking the data class definitions as tagged, i.e.

```
package <base_package>  is
     ---------- Class type declared tagged private ---------------
     type <class> is tagged private;
     .....
---------- Private section of Ada package specification: ---------
private
     [<declaration>]*      ------ Private data definitions
     ------- Class attributes defined in private section: --------
     type <class> is tagged record
           <attribute-1> : <attribute-1_internal_type>;
           ....
           <attribute-n> : <attribute-n_internal_type>;
     end record;
end  <base_package>;
```

Subclasses with additional attributes could use the following template:

```
with <base_package> [,<package>];
package <sub_package>  is
     --------- Class type declared tagged private  ---------------
     type <sub_class> is new <package>.<class> with private;
     .....
-------- Private section of Ada package specification: -----------
private
     [<declaration>]*    ------ Private data definitions
     -------- Class attributes defined in private section: -------
     type <sub_class> is new <package>.<class> with record
           <attribute-j>     : <attribute-j_internal_type>;
           ....
           <attribute-j+k> : <attribute-j+k_internal_type>;
     end record;
```

```
end <sub_package>;
```

When the subclass is a child of the base class, the subclass type declaration uses the base class package, i.e.,

```
type <sub_class> is new <base_package>.<class> with record
```

Otherwise, it uses the package of its immediate parent. Formats for access, update, construct, and destruct operations need not change between Ada 83 and Ada 95. However, alternatives might be preferred in some circumstances in order to better take advantage of Ada 95 features. It may also be preferable to manipulate such objects using an access type as a pointer to better ensure object uniqueness.

One area in which class operations may benefit from new Ada 95 features is in the construction and destruction of objects. Ada 95 defines new tagged types called *Controlled* and *Limited_Controlled* in the library package "Ada.Finalization" that supports operations of *Initialize* and *Finalize* for all objects defined as their derived types. The *Finalize* procedure is automatically called when leaving the scope of a controlled object's declaration in order to reclaim its space. It is designed to automatically handle memory reclamation for difficult cases involving the destruction of access types. *Initialize* is called on every controlled subcomponent of an object which is not assigned an initial value during elaboration of the object's declaration. These are defined as abstract procedures, so that the programmer may tailor them for specific classes to perform any other required activities upon object creation and/or destruction, such as removal of associations [ANSI95, Section 7.6].

4.3 INHERITANCE

Full-featured inheritance is much more straightforward in Ada 95 than it is in Ada 83. We have already discussed how Ada 95's tagged types support inheritance with attribute extension in subclasses. In addition, Ada 95 provides a new hierarchical library structure that supports the inheritance of the private parts of parent packages by the private parts (and bodies) of their children. A child of a package is specified simply by beginning the child package name with the parent package name (e.g., <parent>.<child>). In circumstances where subclasses benefit from visibility into the private parts of their superclasses, inheritance can be implemented using both derived tagged types and these hierarchical library units (HLUs). If such visibility is unnecessary for a particular hierarchy of classes, it may be best to avoid use of HLUs. When a child package is designed so its proper functioning depends on the inner workings of its parent package, the linkages between parent and child are strengthened, reducing their modularity and making independent modifications and

maintenance more difficult.

A simple example template for inheritance in Ada 95, using both the package hierarchy and the type hierarchy, can be given for a child package <parent>.<child> as follows:

```
package <parent>.<child>  is
     ------------ Class type declared tagged private  ---------------
     type <sub_class> is new <parent>.<class> with private;
     .....
    ----- Private section of Ada package specification: --------------
     private
           [<declaration>]*    ------ Any private data definitions
     --------- Class attributes defined in private section: -----------
           type <sub_class> is new <parent>.<class> with record
              <attribute-j>   : <attribute-j_internal_type>;
              ....
              <attribute-j+k> : <attribute-j+k_internal_type>;
           end record;
     end   <parent>.<child>;
```

While Ada 95 provides superb support for single inheritance through the dual features of tagged types and HLUs, it is restricted to only single inheritance in both respects as neither derived types nor child packages may have more than one parent type or package, respectively. Whether single or multiple inheritance is preferable for OO systems remains controversial. While multiple inheritance allows greater flexibility in the inheritance structure, it creates problems when inheritance clashes arise through the inheritance of different operations from different parents.

4.4 CONCLUSIONS

Ada 95 provides excellent support for OO inheritance and polymorphism, rectifying the deficiencies of Ada 83 in these areas.

The new tagged data types of Ada 95 support attribute extension in classes. Polymorphism is supported by the class-wide type variants of tagged types which may be instantiated to any of the classes in an inheritance hierarchy. The new package hierarchies using HLUs support greater visibility of parent classes to their children (subclasses). Once validated compilers are available, Ada 95 will greatly simplify OO implementation in Ada. However, when unvalidated Ada 95 compilers are to be used in the interim on a system, validated compilers should be used prior to final testing of that system.

5. CLASS-WIDE PROGRAMMING IN ADA 83

Achieving a high level of encapsulation and modularity while supporting subclass-specific attributes turns out to be fairly complex in Ada 83. Techniques for achieving such capabilities typically involve unchecked conversions between class datatypes to allow an augmented subclass to look like a base class. Performing unchecked conversions is a risky programming practice that violates Ada's strong typing conventions. The results of such practices can be compiler dependent, although most existing compilers will support it without any problems. Thus implementing this strategy for OO programming in Ada 83 carries risks that some projects may prefer to avoid by using the simpler variant-record strategy described in Chapter 3. But when greater OO capabilities are desired, the Ada literature details a variety of approaches for achieving them, including those described in [BAKR91, SEID92, HIRA92, HIRA94a, HIRA94b].

Here we present the details of an approach that uses unchecked conversion that provides a better combination of modularity and encapsulation than is possible with a variant record approach. A complete, if simplified, example using this strategy is discussed in a companion report [IDA95c] where an Ada 83 OO program is used to wrap legacy Cobol code. The full annotated code of this example is provided in Appendix A.

5.1 ENCAPSULATION

Templates of Ada 83 code are presented in this and following subsections to outline this strategy to OO programming using Ada 83. Missing details in any particular template are either filled in later, such as where the <Attribute Type> is declared below, or referred back to a prior template, as in many of the subsequent templates. A general template illustrating alternative means of encapsulation in Ada is presented:

```
package <Class_Name> is
   type Class is [limited] private;

   -- Object Management:
   -- Description of strategy to avoid memory leaks [including
   -- plan to migrate to Ada 95]
```

```
   procedure Construct  (Object  : in out Class); -- can be a function
   [procedure Assign    (Left    : in out Class;
                         Right   : in     Class);]
   procedure Destruct   (Object  : in out Class);
   -- Attributes access operations:
   procedure Change (Self : in Class;
                <Attribute_Name> : in <Attribute_Type>);
   function Get (Self : in Class) return <Attribute_Type>;
   ...
   -- Operations:
   procedure Operation (Self : in Class{;
      <formal_parameter_definition>}*);
   ...
private
   type Structure is
      record
         (null; | {<Attribute_Name> : <Attribute_Type>;})
      end record;
-- Either reference an object through a pointer,
   type Class is access Structure;
-- or reference the object through an array index,
   Maximum_Number_of_Objects : constant := TBD;
   type Class is range 1..Maximum_Number_of_Objects;
   Pool : array (Class) of Structure;
-- or if no reference is needed, then use limited and no assignment:
   type Class is new Structure; -- or define the record directly
end <Class_Name>;
```

Several conventions are noted:

- The type is called “Class” and is made private or limited private (the trade-offs are beyond the scope of this paper).
- The name of the class is used as the package name.
- The operations are grouped to reflect either what part of the OO design they represent, object management, attribute access, or general operations.
- Construct and Destruct operations use the formal parameter name “object”.
- All other operations use the formal parameter name “Self” to identify the target of the operation. Other parameters should follow an appropriate convention.

- The data structure for object is implemented as a record called “Structure”, and contains a declaration for each attribute or “null” if there are no attributes. This is really an implementation detail for discussion purposes. Other implementations might store attribute values in hash tables or data bases.
- The class implementation is hidden and is also implementation dependent. The three choices illustrated in the template are a pointer, index, or as the structure itself. In the case of pointers and indexes, assignment can be allowed while preserving the semantics of OO programming. If the class is implemented as the record structure, then assignment must be disallowed by declaring the type, Class, limited private. Of course, a procedure that performs assignment can be added and then implemented with the correct behavior.
- Ada programmers may notice an apparent inconsistency in the mode of the parameter Self. Except in the case of construction and destruction, the parameter mode is “in” even though an attribute is being modified. This is possible because the unique index (identifier, address) is always required and should not change. In practice, this allows the programmer to pass a function result to the formal parameter Self, a significant convenience.

There are several decisions for the engineer to make in designing a class interface. The first decision is whether multiple references to an object will be needed. If not, then the difficulties of memory management are greatly simplified and references are not needed. To enforce the single reference semantics, the type Class can be made limited and no assignment operation provided. This is a severe restriction, though, and rarely occurs in OO programming.

If multiple references will be needed (as is usually the case), then the engineer chooses some strategy for allocating and deallocating objects and managing the references to an object. There are many approaches to dealing with this complex problem. Ada 95 introduces the predefined package, Ada.Finalization, and associated semantics to give the engineer some automated assistance. Whatever strategy the engineer chooses for an Ada 83 class interface, a strategy for migrating to Ada 95 should also be chosen.

The template presented in this paper suggests two basic implementations of multiple reference semantics, access type or array index. When an access type is used, the memory needed to store the structure is allocated on a heap by the run-time system when needed or possibly in memory pools. The number of objects is limited by the available heap space.

An array index references an array allocated when the package is first elaborated (at program start-up if the package is a library package). This fixed array puts an upper bound on the number of objects that can exist, allocates all of the space at start-up, etc. A complete discussion of the trade-offs and alternative implementations is beyond the scope of this paper.

5.2 ATTRIBUTES

Although a package specification defines the class interface, encapsulation can be used for other purposes than implementing OO classes. Consequently, not all Ada packages define a class. For example, an attribute may take on values that require non-trivial operations, such as a person's name. Identifying the last, first, middle, title, etc., and maintaining these distinct parts constitute enough complexity to warrant encapsulation. This is commonly referred to as an *abstraction* and will in practice look similar, if not identical, to class encapsulation. There are, however, no requirements involving uniqueness and reference semantics as found in classes. In fact, the behavior should be documented in the package specification to avoid confusion. For example, if a person's name is also the attribute of some roster, will the name change on the roster when the title field of the name is updated?

The following template illustrates how an attribute abstraction is usually defined. Values of type *Attribute* should have copy semantics for assignment. For example, if B is assigned to A, and then B changes value, A retains the previous value.

```
package <Abstraction_Name> is
   type Attribute is private;
   procedure Change
      (A          : in out    Attribute;
       New_Value  : in        <Some_Type>);
   function Get (A : in Attribute) return <Some_Type>;
   ...
private
   type Attribute is TBD;
end <Abstraction_Name>;
```

The conventions used for defining abstract data type are as follows:

- The package is given the abstraction's name.
- The type is named "Attribute". Another good name is "Value", to signify the semantics of assignment.

- Refer to style guides for recommendations on naming conventions for operations and parameters as well as discussions about visible, private, and limited type declarations.

Some attribute values are implemented using the simple predefined types such as Ada 83's Integer, Boolean, String, etc. An enumerated type is another way to define the values of an attribute without adding a separate package definition. Whether to define a separate package for attribute values or declare the values in the class interface is an engineering decision based on the trade-offs of local vs. global complexity, reuse, and planning for change. Adding packages can increase global complexity while providing greater potential for reuse. Declaring the attribute values in a class interface can localize definitions for ease of understanding while complicating reuse. A case can be made in both situations in planning for change and will depend heavily on how the object is used in the application domain.

The original template left out the details of how to deal with attribute values, declared by the <Attribute Type>. The following template illustrates how to deal with attribute values defined in a separate abstraction package and attribute values defined locally (other parts of the class template are ellided):

```
{with <Abstraction_Name_1>;}*
package <Class_Name> is
   type Class is [limited] private;
   {type <Local_Attribute_Type> is
      [<enumeration> | <integer_subtype> | <string_subtype> |...];}
   -- Object Management:
   ...
   -- Attribute access operations:
   -- Use dot notation instead of "use"ing the package:
   procedure Change     (Self : in  Class;
         <Attribute_Name_1>   : in  <Abstraction_Name_1>.Attribute);
   function Get         (Self : in   Class)
      return                        <Abstraction_Name_1>.Attribute>;
   ......
   -- Locally defined attribute values don't need the dot notation.
   procedure Change     (Self : in Class;
       <Attribute_Name_i> : in<Local_Attribute_Type_i>);
   function Get         (Self : in Class)
      return                 <Local_Attribute_Type_i>;
```

```
      .......
      -- Operations:
      ...
   private
      type Structure is
         record
            <Attribute_Name_1> : Abstraction_Name_1>.Attribute;
            .....
            <Attribute_Name_i> : <Local_Attribute_Type_i>;
            .....
         end record;
      ...
   end <Class_Name>;
```

The conventions for implementing class attributes used here are as follows:

- Declare the type of attributes, or include an appropriate package that declares them by using a “with” statement. Those attributes defined in other packages can be renamed with a subtype declaration (not shown). If operations are needed for the locally defined attribute types, a separate package should be used to define the abstraction.
- Provide the change and get operations for each attribute.
- The attribute name is used both as the formal parameter name for the associated operations and as the component name in the record, Structure.

There are alternatives to overloading the operation names. For example, the attribute name can be appended to either or both operation names:

```
procedure Change_<Attribute_Name>
   (Self       : in  Class;
    New_Value  : in  <Attribute_Type>);

function Get_<Attribute_Name>
   (Self       : in  Class)
    return           <Attribute_Type>;
```

The attribute name can be used in place of either operations, although this is more commonly done for the Get operation:

```
function_<Attribute_Name_1>
```

```
(Self       : in  Class)
 return           <Attribute_Type>;
```

All of these alternatives to naming the attribute operations have benefits and difficulties. The operation names are simpler and more uniform without the attribute names, though such treatments may require making the datatypes of an operation's arguments more explicit in order to distinguish the attribute of interest. Changing a name, for example, may require specifying a string constant to be of the intended type:

```
Bobs_Name : constant First_Name := "Bob";
Change(Person, Bobs_Name);
```

This approach also requires that attribute data types be kept distinct. Though this is generally a good programming practice in accord with Ada's strict typing, it may be inconvenient in cases where different attributes must be compared or combined since it may require extra type conversions.

Including the attribute name in the access operation names allows making changes to attribute using constant values whose type might otherwise be ambiguous:

```
Change_First_Name(Person, "Bob");
```

It may also make the code easier to read in some cases, obviating any need to go back and check a datatype in order to disambiguate an operation. However, it may encourage laxity in maintaining distinct types for distinct attributes. Either approach can be adequate in most development contexts. Whichever is chosen, it is helpful to record it in the project style guide and to use it uniformly throughout the system.

5.3 INHERITANCE

Ada 83 provides a restricted form of static inheritance which can work in some cases to implement inheritance found in an OO design. The language construct is called a *derived type* and takes the following form:

```
type <Child_Type> is new <Parent_Type>;
```

All of the operations that work on the parent type are inherited and new operations can be added or inherited operations can be overridden. However, the child type cannot extend its data structure to hold additional attribute values. This limitation has been a significant difficulty for programmers trying to implement inheritance in Ada 83. Ada 95 addresses this limitation by introducing class wide programming for types declared as **tagged**.

The strategy presented in this paper addresses the interrelated issues of inheritance and polymorphism. Some of these implementation issues are discussed in [JHNS93, BAKR91, HIRA92].

Looking toward the Ada language revision, this paper recommends a variant record to implement the data structure of the base class. The discriminant implements the “tag” which will be replaced when upgraded to Ada 95. The naming convention ensures that an Ada 95 compiler will identify the illegal use of the reserved word, **tagged**, for the programmer to fix. This implementation differs from the common use of variant records to achieve polymorphism by maintaining a modular package structure and ignoring the case construct when defining the variant record structure. The following template highlights how a base class is implemented so that other classes can be derived and the data structure extended:

```
package <Parent_Name> is
   type Class is private;

   -- Object Management:
   ... -- See other templates

   -- Attribute access operations:
   ... -- See other templates

   -- Operations:
   ... -- See other templates

-- For Child packages only (see Ada 95):
-- Start private section here when migrating to Ada 95!
   type Tag is (<enumeration> | <natural subtype>);
   type Structure (Tagged : Tag) is private;

   -- If objects of this parent class are constructed, then use
   -- one of the Tag values. This is not necessary in Ada 95,
   -- so delete this declaration when migrating.
   [Unique_Tag : constant Tag := TBD;]

private
   type Structure (Tagged : Tag) is
      record
         Attribute_1 : Attribute_Type_1;
```

```
            ... -- No case statement here!
         end record;
      type Class is access Structure; --'Class (in Ada 95)
   end <Parent_Name>;
```

The conventions highlighted by this template are as follows:

- Using the identifier, **Tagged**, will cause an Ada 95 compiler error message. This will help when the code is upgraded to Ada 95 to convert all occurrences of these variant records to tagged record types.
- The types Tag and Structure are declared just before the private declarations. They are intended for the sole use of child packages (as described below). However, there is no way to enforce the desired visibility in Ada 83. When upgrading to Ada 95, the type Tag is removed, and the type Structure is moved to the private section which will achieve the desired visibility.

The descendant classes can now inherit the parent operations as described earlier, and add or override operations as needed. The structure is extended using a unique tag to identify at run time the form of an object (i.e., polymorphism). This template highlights how a descendent class is declared:

```
with <Parent_Name>;  -- not needed in Ada 95

-- For the Ada 95 package name, replace the underscore with a
-- period.
package <Parent_Name>_<Child_Name> is
   type Class is new <Parent_Name>.Class;

   -- Class management operations must be overridden:
   procedure Construct  (Object  : in out Class); -- can be a function
   [procedure Assign    (Left    : in out Class;
                         Right   : in     Class);]
   procedure Destruct   (Object  : in out Class);

   -- New attribute access operations:
   function    Get...;
   procedure   Change...;
   ...
```

```
   -- New operations:
   procedure Operation (Self : in Class{;
      <formal_parameter_definition>}*);
   ...

-- This Unique_Tag is used for dispatching, See <Parent Name>.Abstract
-- In Ada 95, remove this declaration.
   Unique_Tag : constant <Parent_Name>.Tag := <Unique_Value>;

private
   type Structure is -- new <Parent_Name>.Structure with (Ada 95)
      record
         Parent   : <Parent_Name>.Structure(Unique_Tag);
         Att1     :...; -- New Attribute
         ...
      end record;
   -- In Ada 95, some additional operations may be needed if
   -- Structure is an abstract type, such as
   -- Ada.Finalization.Controlled.
end <Parent_Name>_<Child_Name>;
```

The conventions for declaring a descendent class are as follows:

- As suggested by the Ada 95 syntax for child packages, the parent and child class names are catenated, separated with an underscore instead of a period.
- Of course, the parent class is made visible using a context clause and as described earlier, the type is derived in order to inherit the parent operations.
- Every class that constructs an object must have a tag value associated with that identifies to which class the object belongs. The identifier Unique_Tag is used in this template and declared immediately before the private declarations to indicate it is for the use of the class hierarchy. It should always be constant and its value unique through some mechanism.
- The type Structure is defined as needed to extend the parent data structure with the new attributes. Since this is implementation dependent, so are the conventions. “Consistency” is the best guide.

There can be as many child (i.e., descendent) classes as there are unique tag values. This leads to an obvious design decision in Ada 83 for the parent class definition: Should

the tag be enumerated or natural? If the tag is enumerated, descriptive names can be used to help maintenance and identify mistakes. If the tag is a natural (or subtype), there is virtually no limit to the number of children that can be added without modifying the parent package. This is again an engineering decision made by the implementor on a case-by-case basis. The example in this paper uses an enumerated type.

There are benefits and problems with this approach to extending a data structure. The most obvious benefit is its similarity to Ada 95 constructs: the use of hierarchical "child" packages will obviate the need to make the structure declaration visible. The child packages' names will use the "." (period) delimiter instead of "_" (underscore) and consequently remove the context clause (**with**). Pointers will continue to be the most common way to maintain object uniqueness and reference semantics. The private type declarations and type derivation will continue to be the most common way to implement encapsulation and inheritance.

Other benefits include the encapsulation and inheritance implementation is almost identical to the design model, and the implementation details, including data structures, are almost completely hidden.

There is a drawback with this approach that will be remedied by Ada 95: this approach is (compiler) implementation dependent. Normally, this would be a serious problem, but in practice, the implementation presented here will work for most compilers or only slight changes will be needed.

Before showing the package body template, a short diversion is helpful. In Ada 95, an attempt to operate on an object with the wrong tag raises an exception, Ada.Tags.Tag_Error. To simulate this in Ada 83 with minimal effect to the code during migration to Ada 95, a package, Ada, can be defined using the Ada 95 specification. Other operations found in the Ada 95 specification can be implemented if needed, and then discarded when migrating from Ada 83:

```
package Ada is -- See Ada 95
   ...
   package Tags is
      ...
      Tag_Error : exception;
   end Tags;
   ...
end Ada;
```

The implementation presented here introduces a compiler dependency when a pointer to the child structure is used in place of a pointer to the parent structure. In practice, this does not cause a problem, so that the child package can construct the larger structure to hold both the parent attributes and the additional child attributes as illustrated in Figure 1. The body of the child package has a pointer to the larger structure for its own use, con-

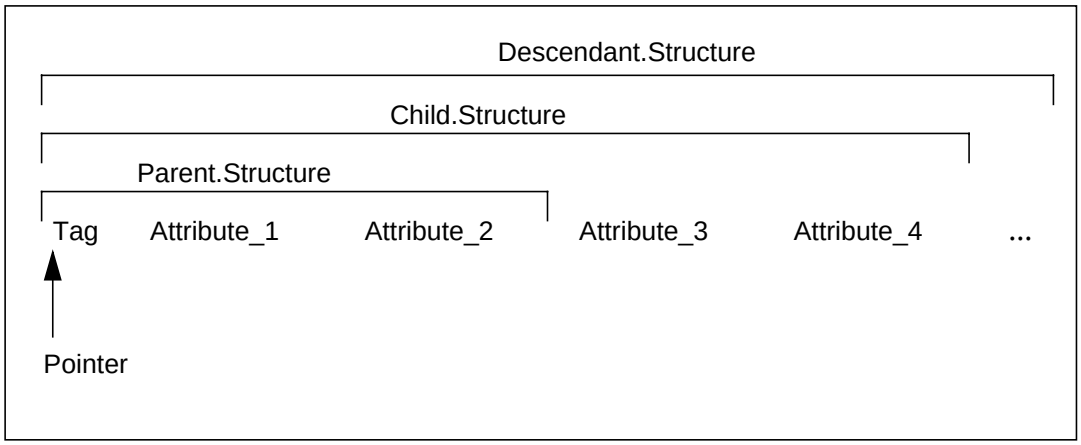

Figure 1. Data Structure Extension in Ada 83

verting back and forth to the visible pointer as needed. This template shows an implementation used in the example in Appendix A:

```
package body <Parent_Name>_<Child_Name> is
   -- Allocate pointers to the larger child structure, then
   -- treat it as a pointer to the parent structure until any
   -- operations on the child attributes are used.
   type Child_Pointer is access Structure; -- of the child
   -- Provide a type safe conversion utility to access all
   -- of the new attributes:
   function Narrow (Parent_Pointer : in Class)
      return Child_Pointer is separate;
   -- Implement the constructor to allocate an entire
   -- Child_Structure!
   procedure Construct  (Object  : in out Class)
      is separate;
   -- other Object Management operations
   ...
   -- Use the function, Narrow, for the rest of the operations:
   ...
   -- New attribute operations:
```

```
   function Get (Self : in Class) return <Attribute_Type> is
   begin
      return Narrow(Self).<Attribute_Name>;
   end Get;
   ...
   -- Other new operations:
   ...
end <Parent_Name>_<Child_Name>;
```

The conventions introduced by this template are as follows:

- A pointer to the extended child structure is declared. The exact definition of the type Child_Pointer and the implementation of the Narrow function and the object management operations are implementation dependent.
- The Narrow function is declared to handle conversion from the general class which is a pointer to the parent structure to the access type, Child_Pointer, which is a pointer to the child structure. (See the implementation of the Narrow function in the next code example.)
- The Narrow function is used by all of the operations whenever they need to update or retrieve the value of an attribute added by the child class.

The Narrow function provides a way to see the attributes added to the child structure. By going through this one routine, the required check on the tag can be implemented in this one routine, as well as any implementation dependent code:

```
with Unchecked_Conversion; -- Be careful
with Ada; -- Tags
separate (<Parent_Name>_<Child_Name>)
function Narrow (Parent_Pointer : in Class)
   return Child_Pointer is
   function Convert_Pointer is
      new Unchecked_Conversion
         (Source => Class,
          Target => Child_Pointer);
   Result   : constant Child_Pointer
            := Convert_Pointer (Parent_Pointer);
begin
   if Result.Parent.Tagged = Unique_Tag then
      return Result;
```

```
      else
         raise Ada.Tags.Tag_Error;
      end if;
   end Narrow;
```

The conventions for this template are as follows:

- If not already visible, Unchecked_Conversion and Ada are named in a context clause.
- The unchecked conversion from the Parent_Pointer to the Child_Pointer is performed first in order to see the value of the discriminant, Tagged.
- As suggested by the Ada 95 language, the exception Ada.Tags.Tag_Error is raised if object has the wrong tag, indicating that the object was not initialized using the operation in this package.

The function, *Construct*, allocates a child structure and converts the pointer to the visible pointer, Class:

```
with Unchecked_Conversion; -- Be careful
separate (<Parent_Name>_<Child_Name>)
procedure Construct    (Object: in out Class) is
     function Convert_Pointer is
           new Unchecked_Conversion (Child_Pointer, Class);
begin
     Object := Convert_Pointer (new Child_Structure);
end Construct;
```

No particular convention other than a correct implementation is needed here.

The implementation outlined here provides a type safe approach. The engineer is required to follow these rules when implementing the child classes:

- Check at run time whether the object Self is of the correct type by always evaluating the tag. In the template, this is accomplished by always using the Narrow utility which raises an exception when any attempt is made to convert (narrow) an object with the wrong tag.
- Every derived type (child class) must have a unique tag. There are many ways to accomplish this. The unique tag may even be allocated at run time.

- The Child constructor must allocate the data structure in such a way that the compiler can use the pointer in both the parent and child package. This is a compiler dependency.

In practice, it is difficult to create a run-time type mismatch. Only if the engineer forces a type conversion and then invokes one of the child operations will the exception be raised. In the following code fragment, the operation Get will raise the exception Ada.Tags.Tag_Error:

```
...
   Par_Object        : Parent.Class;
   Child_Attribute   : Parent_Child.Att;
begin
   Parent.Construct (Par_Object);
   Child_Attribute := Parent_Child.Get
      (Parent_Child.Class (Par_Object)); -- Type Conversion!
   ...
end;
```

5.4 POLYMORPHISM

Dispatching of polymorphic operations can be accomplished in two ways which mirror the semantics of Ada 95. When a child class overrides an operation, the parent class must dispatch to the child operation:

```
package <Parent_Name> is
   -- See other templates
   ....
   procedure <Overridden_Operation> (Self : in Class{;
      <Formal_Parameters>});
   ...
end <Parent_Name>;

with <Parent_Name>_<Child1_Name>; -- To dispatch
with <Parent_Name>_<Child2_Name>; -- To dispatch
....
package body <Parent_Name> is
   ...
   procedure <Overridden_Operation> (Self : in Class{;
      <Formal_Parameters>}) is
      package Child1 renames <Parent_Name>_<Child1_Name>;
```

```
      package Child2 renames <Parent_Name>_<Child2_Name>;
      ....
   begin
      case Self.Tagged is -- Was set in Construct
         when Child1.Unique_Tag =>
            Child1.<Overridden_Operation>
               (Child1.Class(Self){, <Formal_Parameters>});
         when Child2.Unique_Tag =>
            Child2.<Overridden_Operation>
               (Child2.Class(Self){, <Formal_Parameters>});

         when others => -- Default behavior
            ...; -- Perform parent behavior

      end case;
   end <Overridden_Operation>;
   ...
end <Parent_Name>;
```

There are some conventions used here:

- If long names become a problem, the child package can be renamed as opposed to naming it in a “use” clause.
- A case statement includes an “others” clause only when there is default behavior associated with the operation. The lack of any default behavior is a clue that this is possibly an abstract operation and should be moved to the embedded package, abstract.
- Overridden operations can be compiled separately, which would allow the context clauses to be removed from the package body. This is probably more helpful when only a few operations are overridden.
- The structure of the case statement, the invocation of the child operation, and the type conversions are dictated by the conventions used in other templates and the Ada 83 language definition.

Dispatching also happens when a parent class defines an operation as abstract, meaning that every child class must implement that operation and the abstract operation will always dispatch to the appropriate child operation. Every child type must provide a

concrete definition of the parent's abstract operations. The following template shows how abstract operations can be declared:

```
package <Parent_Name> is
   type Tag...
   type Structure (Tagged : Tag) is private;
   type Class is private;

   ... -- See other templates

   package Abstract is -- All abstract operations
      procedure <Operation_Name> (Self : in Class{;
                                    <parameter>}*);
      ...
   end Abstract;
private
   ...
end <Parent_Name>;
```

The convention associated with declaring abstract operations is when the Ada 95 keyword *abstract* is used as the package name to hold the declaration of the abstract operations. When upgrading to Ada 95, package declaration can be removed and the operations declared as abstract using the appropriate syntax.

The abstract package convention identifies which operations are abstract and must be defined by the child packages. The following template shows how the descendent packages declare and implement the abstract operations:

```
with <Parent_Name>;

package <Parent_Name>_<Child_Name> is
   type Class is new <Parent_Name>.Class;

   ...
   -- Define all parent abstract operations as concrete:
   procedure <Operation_Name> (Self : in Class{;
                                 <parameter>}*);
   ...
private
   ...
```

```
end <Parent_Name>_<Child_Name>;

package body <Parent_Name>_<Child_Name> is
   ...
   procedure <Operation_Name> (Self : in Class{;
                                  <parameter>}*) is
   begin
      ... -- Behavior for Child objects
   end;
   ...
end <Parent Name>_<Child Name>;
```

The rules of Ada 83 require the parameter profiles to match. Ada 95 will also check that all abstract operations have the corresponding concrete declarations. The only convention provided in this approach which helps identify whether all of the abstract operations have been declared is to implement the body of the abstract operation.

When the concrete operations are implemented, the programmer must be careful not to recursively invoke the parent's abstract operation (this is true for all dispatching operations). In Ada 95, the abstract operations dispatch to the appropriate concrete operation using the tagged discriminant. In Ada 83, this must be programmed explicitly:

```
package body <Parent_Name> is
   ...   -- See other templates,
         -- including dispatching on overridden operations

   -- Separating the package, Abstract, is optional.
   package body Abstract is separate;

end <Parent Name>;
```

And in a separate file, the package body Abstract is defined:

```
with <Parent_Name>_<Child1_Name>; -- To dispatch
with <Parent_Name>_<Child2_Name>; -- To dispatch
... -- other children
separate (<Parent_Name>)
package body Abstract is
   procedure <Operation_Name> (Self : in Class{;
                                  <parameter>}*) is
   ...
```

```
      package Child1 renames <Parent_Name>_<Child1_Name>;
      package Child2 renames <Parent_Name>_<Child2_Name>;
   begin -- Dispatching
      case Self.Tagged is -- Was set in Construct
         when Child1.Unique_Tag =>
            Child1.<Overridden_Operation>
               (Child1.Class(Self){, <Formal_Parameters>});
         when Child2.Unique_Tag =>
            Child2.<Overridden_Operation>
               (Child2.Class(Self){, <Formal_Parameters>});
         ... -- other children
         -- However, no others clause! This is an abstract operation!
      end case;
   end <Operation_Name>;

   ...
end Abstract;
```

Note the following conventions:

- The package Abstract is defined separately. This moves the context clauses to the separate package and simplifies the parent's package body.
- Each abstract operation implements a case statement on the tag, dispatching to the correct child operation.
- Because the operations are abstract, no "others" clause should be added (although it may be necessary if all of the tag values cannot be enumerated such as an integer tag; in which case, an exception should be raised such as Program_Error).

Note the separation of responsibilities. The definition of how to implement the abstract operation is defined concretely within the child class definition. The abstract operation itself is written to be a simple dispatching mechanism in the parent class.

When there is the need to override a parent operation, Ada 95 semantics require that the corresponding operation on the class-wide type dispatch to the overridden operation. There is no similar class-wide type in Ada 83. The approach used in this paper is to use the parent type to implement run-time dispatching. Consequently, it is possible to treat the parent type as a class-wide type and implement the overridden operations to dispatch. This

again makes the dispatching semantics explicit and requires the engineer to maintain the semantic contract between the parent and child.

The promise of OO technology can be described more concretely in terms of maintenance. Fixes in the implementation of child classes are more likely isolated to the body of the child packages. The addition of child classes to enhance a program's capability is less likely to require changes in the other classes in the hierarchy. The Ada 83 approach described here makes the changes to the parent essentially mechanical and isolated to simulating the required dispatching behavior.

An example (still in template form) shows how dispatching can be exploited. An array consisting of different kinds of objects can be populated, and then operations which dispatch can be invoked without concern for the child specific implementations:

```
with <Parent_Name>;
procedure Illustrate is
   List : array (1..10) of <Parent_Name>.Class;
   procedure Populate_List is separate;
begin
   Populate_List;
   Cause_Dispatching:
   for Object in List'Range loop
      -- Invoke an abstract operation:
      <Parent_Name>.Abstract.<Operation_Name>
         (Self => List(Object),...);
      -- Invoke an overridden operation:
      <Parent_Name>.<Overridden_Operation>
         (Self => List(Object),...);
   end loop Cause_Dispatching;

end Illustrate;
```

An array (*List*) is specified to take objects of the type of the base class as values. When objects are entered into the array, however, they may be constructed to be any of the subclasses of the base class. In this example, a separate procedure, Populate_List, is used to fill the array with values. After the array is populated, a loop for dispatching is implemented that illustrates invocation of dispatching operations both directly from the parent package and from an abstract package within the parent. In every case, an object from the

list is supplied as principal argument to the operation, supplemented by whatever other arguments are required.

```
with <Parent_Name>_<Child_Name_1>;
with <Parent_Name>_<Child_Name_2>;
separate (Illustrate)
procedure Populate_List is
begin
   for Object in List'Range loop
      -- Use some criteria for populating the array with
      -- different objects:
      if (I rem 2) = 0 then
         <Parent_Name>_<Child_Name_1>.Construct
            (<Parent_Name>_<Child_Name_1>.Class(List(Object)));
      else -- (I rem 2) = 1
         <Parent_Name>_<Child_Name_2>.Construct
            (<Parent_Name>_<Child_Name_2>.Class(List(Object)));
      end if;
   end loop Populate List;
end Populate_List;
```

Since the construct operations of different subclasses are called in this program unit, their packages must be included (with'd) as indicated. In this template, a loop is used to populate the array with one type of child for odd-numbered array elements and another for even numbered elements. Ordinarily, some other criteria, such as the tags of objects entered from a terminal or obtained from a database, would be used to provide a more meaningful basis for choosing the subclass of the constructed class. Notice that each argument to a construction operation must convert an array object to a matching data type for the class being constructed. This conversion does not change the type of the array elements; it merely changes how they appear when called as arguments to the operation *<Parent_Name>_<Child_Name>.Construct*. The construction itself changes the contents of what is being pointed to but does not change its type in the array. Subsequent dispatching operations will, of course, identify the appropriate type and dispatch accordingly, based on the tag in each object structure.

5.5 CONCLUSIONS

The class-wide strategy for OO programming in Ada 83 achieves a higher combined degree of modularity and encapsulation than the variant record strategy. However, the class-wide strategy still depends on case statements in the parent class to dispatch operations on class-wide types to the appropriate subclass— hence violating modularity in this regard. It is also a riskier strategy due to the use of unchecked conversion, though this risk is mitigated by restrictions on such conversions.

The examples given previously illustrate the greater modularity and encapsulation achievable with this class-wide strategy for OO programming in Ada 83. The class-wide strategy uses unchecked conversion between different datatypes in order to allow a subclass to have a different data structure from its parent, consisting of the parent data structure augmented with additional attributes. This strategy provides full encapsulation of each class-subclass structure definition within the private parts of its package and allows all subclass operations to be defined in their associated subclass packages, providing a combination of encapsulation and modularity not possible with the variant-record strategy.

Polymorphism, however, still requires the use of case statements to dispatch operations on class-wide types to the appropriate subclass operation. Thus, addition of new subclass operations is not fully modular since any such changes are not local to the subclass definition but must also be reflected in the (parent) package in which dispatching occurs. This use of centralized, non-modular dispatching is difficult (if not impossible) to avoid when implementing polymorphism in Ada 83, though it might be automated using a precompiler. Thus, the class-wide strategy does as well as can be expected in its implementation of polymorphism.

The two shortcomings of the class-wide strategy are its complexity and its violation of Ada's strict typing by unchecked type conversions. The complexity of this strategy may be a barrier for inexperienced Ada programmers. Implementation and use of the functions *Narrow* and *Convert_Pointer* for type conversions are potential sources of confusion and error. The use of unchecked conversions naturally raises concern since it violates the strict type checking that is a cornerstone of the Ada language approach to more reliable programming. However, unchecked conversion is strictly limited in the class schemes and should not create problems if the conventions detailed above are faithfully followed. In short, the shortcomings of this OO implementation strategy are minimized with experienced Ada programmers and careful code reviews. Thus, the class-wide strategy is recommended for

its greater encapsulation and modularity when an Ada coding environment supports such risk minimization.

6. IMPLEMENTING ASSOCIATIONS

Associations between classes or between specific objects have not yet been covered in this discussion of OO programming in Ada except for the special case of subclass-superclass associations in inheritance. There are no special constructs in either Ada 83 or Ada 95 for creating general associations between classes or their instances. Associations must, therefore, be programmed in whenever Ada is used to implement an OO system. The characteristics of an association and the way it is used in an application affect its design and implementation. Cardinalities greater than one and multiple directions of traversal, in particular, create special challenges for implementation in Ada. In this chapter, alternative strategies for implementing associations in Ada will be described. The basic strategies described are applicable to either Ada 83 or Ada 95, although we use example code and templates in Ada 83 because of the greater availability of Ada 83 compilers at this writing. Comparable examples can be generated utilizing the Ada 95 OO constructs by simply substituting the general Ada 95 class templates (presented in Chapter 3) for the Ada 83 templates used in the following sections.

Although an association in an object model does not imply how it will be traversed, the name chosen for an association does have a meaning which gives an indication of how the related objects participate in the association. For example, the classes, Paycheck and Employee, are related by the association Pays. The choice of the name, *Pays*, instead of *Is_payed_by*, implies that the source (subject) of the association is Paycheck and the target (object) of the association is Employee. Therefore, Paychecks Pay Employees, not Employees Pay Paychecks. In some object models, two names are given, one for each direction.

The meaning of names chosen for associations may suggest the primary direction that an association will be traversed. However, an association name is not a reliable way to make design decisions. An analysis of the behavior for each object will indicate whether the association is traversed in one or both directions. For example, if one of the methods for the *Employee* class needs to verify that the employee has been payed by checking for an associated paycheck, then the association *Pays* needs to be traversed in the “backward” direction. There is usually an alternative name to use for the other direction, such as

Is_Payed_By. The distinction between *source object* and *target object* is made throughout this section to help explain design strategies for each association. However, the terms source and target only make sense when dealing with an association and should not be used as a general naming convention for classes.

6.1 INDEPENDENT OR INTEGRATED

The engineer makes a basic choice to implement the association as an independent data structure or to integrate the association with the object structures. Figure 2 shows simple cases of the two forms. As an independent data structure, an association is a set of pairs which is easily recognizable as a relation. Integrated with the object structures, the association looks more like an attribute of the objects and will gain the corresponding performance benefits.

When an association is integrated with the object structures, a component is added to each structure to handle either the forward or backward traversal of the association. There is no separate data structure, and traversal is handled by following the pointers found in the object structures, as illustrated by the arrows shown in Figure 2. Unfortunately, when both classes of an association have mutual pointers, they must be in the same package in order to have access to each other, as explained in the next section.

When an association is implemented as an independent structure, the object structures are left unmodified. Instead, some structure is built with associated operations to look up an association record for an object and then follow the corresponding pointer. In Figure 2, an array is used for the association structure, though other data structures could be used, as discussed in the next section.

Both the independent and integrated approaches have their respective design and implementation difficulties. The integrated approach is generally faster during traversal; however, it is very difficult to design the classes as separate packages. The independent approach is slower and more complicated to implement; however, it does allow each object and association to be designed as separate packages. As other design issues are examined, this report uses whichever approach is easier to illustrate.

6.2 DIRECTION OF TRAVERSAL

Associations in an object model do not have direction. However, when used in a design, the associations are intended to be traversed during execution of the methods of an object. The directions of traversal that are required determine the options available to the

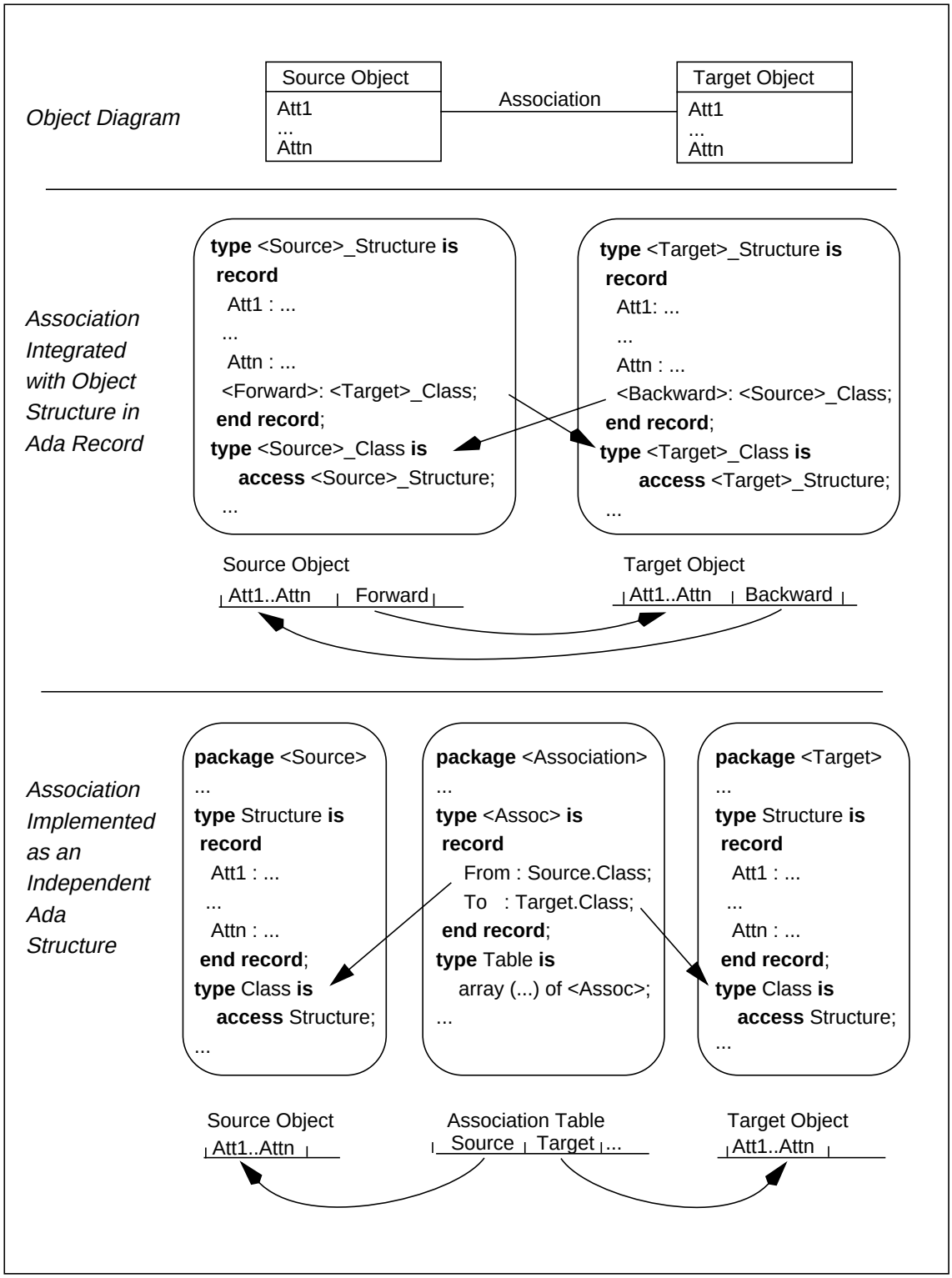

Figure 2. Basic Alternatives in Association Implementation

designer. If an application traverses an association in only one direction, the related objects can be designed so that the object, the source of the traversal, implements the relationship.

For example, a paycheck pays an employee. In the application which writes checks, the pays relation is traversed from a paycheck object to an employee object. Assuming we have no reason to go from an employee object to the paycheck object, this allows the paycheck object to include the relation while the employee object does nothing with the relation, as in the following:

```
package Employee is
   type Class is private;
   ...
end Employee;

with Employee;
package Paycheck is
   type Class is private;
   ...
private
   type Structure is
      record
         ...
         Pays : Employee.Class; -- Integrate association
      end record;
   type Class is access Structure;
end Paycheck;
```

This is the simplest implementation and preserves strong typing, encapsulation, and modularity. Integration occurs within the record *Structure*, where the component *Pays* is used to point to the associated *Employee.Class*. The rest of the object structure is unchanged and if the association needs to be set or traversed by other packages, appropriate operations are added to the class specification.

In Ada, the basic structure of modularity is the package. Packages are made visible to each other through context clauses (**with**'ing). Ada does not allow circular **with** clauses among package specifications—specifically, two packages cannot "**with**" each other. Therefore, when implementing an association between two classes, the engineer cannot simply add a component to each object structure pointing to the object in the other package since this would require circular **with**'ing. Thus, when associations must be traversed in multiple directions, the implementation must take an alternative approach to making each class involved accessible to the other(s). This leads to some design choices which make

trade-offs between modularity and integrating data structures, the principal alternatives being as follows:

- Merging Packages,
- Partial Integration, or
- Independent Association Package.

When class definitions are merged into the same package, then each class definition is visible to the other without the need for any **with**'ing, though this makes them inseparable and could hamper the separate reuse of any of the merged classes. A partial integration of the association with its classes places all the association references in one class, providing greater modularity at the costs of favoring one direction of traversal over the other. An independent package for the association provides equal treatment of traversals and maximal modularity, though it complicates traversal. Each of these alternatives is examined in greater detail in the next section.

6.3 MERGING PACKAGES

When the objects are merged into a single package, the association can be integrated with the object structures. This breaks the modularity of the design for the objects that participate in the association. However, it maintains encapsulation and provides a simple interface for traversing the association:

```
package <Merged> is
   type <Source>_Class is private;
   type <Target>_Class is private;

   procedure Construct (Self_S : in out <Source>_Class;
                        Self_T : in out <Target>_Class);

   function <Forward_Name> (Self : in <Source>_Class)
      return <Target>_Class;
   function <Backward_Name>(Self : in <Target>_Class)
      return <Source>_Class;
   -- ...

private
   type <Source>_Structure; -- Forward declare
   type <Target>_Structure; -- Forward declare

   type <Source>_Class is access <Source>_Structure;
   type <Target>_Class is access <Target>_Structure;
```

```
   type <Source>_Structure is
      record
         -- ...
         <Forward_Name> : <Target>_Class;
      end record;

   type <Target>_Structure is
      record
         -- ...
         <Backward_Name> : <Source>_Class;
      end record;
end Merged;
```

Here are some features of the code to note:

- Since both classes are defined in the same package, there is no need for context clauses between the two classes.
- Optionally, the Construct operation can construct both objects and establish the association between them. This is useful for enforcing required associations. If separate construct operations are visible and the association is also established with a separate procedure, then it is up to the user of the package to ensure that the association is maintained with the correct cardinality.
- A naming convention is introduced to distinguish between the two type declarations for each class.
- In the private part of the specification, the structure types are declared before the definition (forward declaration), so that the classes can be defined as pointers to their respective structures. The completed class type definitions can then be used to complete the structure definitions. The building of these structure definitions cannot be split across packages without violating other principles of software engineering.
- There can be more associations, but the forward and backward names should be unique in the package. Otherwise, some care must be taken in overloading the operations.

The implementation of the merged package is straightforward and is shown here for comparison with the alternative designs shown in subsequent sections:

```
package body Merged is

   procedure Construct (Self_S : in out <Source>_Class;
                        Self_T : in out <Target>_Class) is
   begin
      Self_S := new <Source>_Structure; -- Construct Source
      Self_T := new <Target>_Structure; -- Construct Target
      -- Establish the association:
      Self_S.<Forward_Name>  := Self_T;
      Self_T.<Backward_Name> := Self_S;
   end;

   function <Forward_Name> (Self : in <Source>_Class)
      return <Target>_Class is
   begin
      return Self.<Forward_Name>;
   end;

   function <Backward_Name> (Self : in <Target>_Class)
      return <Source>_Class is
   begin
      return Self.<Backward_Name>;
   end;

   -- ...

end Merged;
```

The implementation of the merged package has certain characteristics:

- In this case, both objects are constructed in the same *Construct* operation. The components *Forward* and *Backward* are set in their respective structures, ensuring that this required association is always present. If this were not a required association, separate *Construct* operations and a procedure to "Establish" the association would appear in the package specification and the code separated as reflected in the comments.
- The forward and backward traversals are extremely simple and efficient.

6.4 PARTIAL INTEGRATION

The engineer may choose one of the objects to integrate with the association. This allows the association to be partially integrated with the object structure and enhances performance when traversing the association from the chosen object. A separate data structure

is used to perform the other traversal. The choice of object is based on performance or can be intuitive. Users of the package specification will expect the traversal operations to go in a logical place, even if there is no performance advantage.

```
with <Target>;
package <Source> is
   type Class is private;
   -- <explanation of Construct behavior>
   procedure Construct (Self   : in out Class[;
                        Target : in     <Target>.Class]);
   -- ...

   [procedure Establish_<Forward_Name>
      (Source : in Class;
       Target : in <Target>.Class);]

   function <Forward_Name>  (Self : in Class)
      return <Target>.Class;
   function <Backward_Name> (Self : in <Target>.Class)
      return Class;
private
   type Structure is
      record
         ...
         <Forward_Name> : <Target>.Class;
      end record;
   type Class is access Structure;
end <Source>;
```

Here are some features of this specification to note:

- The package encapsulates only one of the classes. However, it also includes all of the operations for the association.
- The operation Construct can do several things at once. If desired, it can construct the Target object, receive it as a parameter, or do nothing with Target objects. If a Target object is available, the association can be established within the operation Construct immediately after the source object is constructed.
- Depending on what the Construct operation does, a procedure to establish the relation can be provided. The Construct and Establish operations should work together to satisfy the behavior specified by the object model.

- Traversal of the association in the forward direction is accomplished by integrating a reference to the target object within the source structure. The name of the component can be the same as the name of the association, <Forward_Name>.

The package body must provide a way to traverse both directions of the association:

```
with Hash_Table; -- To handle backward traversal of association [it?]
package body <Source> is
   package <Backward_Name>_Table is
      new Hash_Table (Target.Class, Class);

   procedure Construct (Self   : in out Class[;
                        Target : in     <Target>.Class]) is
      ...

   procedure Establish_<Forward_Name>
      (Source : in Class;
       Target : in <Target>.Class);
   begin
      Source.<Forward_Name> := Target;
      <Backward_Name>_Table.Add (Target, Source);
   end;

   function <Forward_Name>  (Self : in Class)
      return <Target>.Class
   begin
      return Self.<Forward>;
   end;

   function <Backward_Name> (Self : in <Target>.Class)
      return Class is
   begin
      return <Backward_Name>_Table.Lookup (Self);
   end;

end <Source>;
```

Some features of this package body to note:

- The backward traversal is handled with a hash table utility. This utility is built in the package body—in this case, using a generic which is instantiated using the source and object class types. Notice that the hash table goes from the target objects back to the source object.

- When an association is established (either in the Construct operation or the Establish operation), the forward reference is established by updating the source object's value. The backward reference is established by adding the source-object pair into the hash table. This is why the technique is called *Partial Integration*—only one of the traversals is integrated into the object structure.
- The forward traversal is implemented differently than the backward traversal.

This approach maintains the modularity and encapsulation of the objects. The complexity is increased for one of the packages which also enjoys a performance advantage. The other package is completely untouched, simplifying maintenance.

6.5 INDEPENDENT ASSOCIATION PACKAGE

The engineer may use a separate package to implement the association as an independent data structure. This is the most general design and does not effect the object structures. This template illustrates an independent one-to-one association:

```
with <Source Name>;
with <Target Name>;
package <Association Name> is

   procedure Establish (From : in     <Source Name>.Class;
                        To   : in     <Target Name>.Class);
   procedure Remove    (From : in     <Source Name>.Class;
                        To   : in     <Target Name>.Class);

   function <Forward_Name>  (From : in     <Source Name>.Class)
      return <Target Name>.Class;
   function <Backward_Name> (To   : in     <Target Name>.Class)
      return <Source Name>.Class;

end <Association Name>;
```

Some features of this example to note:

- The participating classes are made visible through context clauses (**with**'ing).

- The template suggests names such as Establish and Remove. There are a variety of intuitive names that could be used. However, the names used in class definitions (Construct and Destroy) should be avoided to keep the reader from confusing an association with a class.
- The data structure to implement the association is not visible. The implementation determines what data structure is needed and should be completely hidden in the package body.

The package body of these independent structures could use one of several implementations such as linear search, hash table, binary search, etc., which have varying degrees of complexity and performance characteristics. For example, in the Booch components, one of the map packages can provide the desired behavior for a simple mapping [BOO87, p. 212].

6.6 CONDITIONAL OR REQUIRED

If an association is conditional, the designer provides an operation to establish the association. This allows the participating objects to be constructed independently and then associated at an appropriate time in the processing. For example, both the check and the employee may be constructed separately; then when a payment is approved, the check is associated with the employee:

```
with Check;
with Employee;
package Pays is
   procedure Establish (Source : in [out] Check.Class;
                        Target : in       Employee.Class);
   ...
end Pays;

-- Somewhere in the application processing:
   ...
   Employee.Construct (Current_Employee);
   ...
-- Still later:
   Check.Construct (New_Paycheck);
   Check.Set_Amount (Employee.Calculate_Pay (Current_Employee));
   if Accountant.Approve (New_Paycheck) then
      Pays.Establish (New_Paycheck, Current_Employee);
   else
   ...
```

Some notes about the code:

- The code that uses the class and association packages is most likely located in other class operations.
- This example uses an independent structure to implement the association. However, the same decision must be made when the association is integrated with the class structures.

If the association is required, one of the participating objects can be provided as a parameter to the Construct operation for the other object. For example, if the designer wants to ensure that checks are only constructed when there is an associated time sheet, the construction of a paycheck can require a time-sheet object:

```
with Time_Sheet;
package Paycheck is
   type Class is private;
   procedure Construct (Self        : in out Class;
                        Time_Record : in     Time_Sheet.Class);
   ...
end Paycheck;
```

Another alternative for a required association is to construct the target object when the source object is constructed:

```
package Paycheck is
   type Class is private;
   procedure Construct (Self        : in out Class);
   ...
end Paycheck;

with Time_Sheet;
with Pays;
package body Paycheck is
   ...
   procedure Construct (Self        : in out Class) is
      Time_Record : Time_Sheet.Class;
   begin
      Time_Sheet.Construct (Time_Record);
      Self := new Structure;
      Pays.Establish (Self, Time_Record);
   end Construct;
   ...
end Paycheck;
```

Some notes about this example:

- The Time_Sheet package may be **with**'d in the Paycheck specification if other operations need to declare parameters of the type Time_Sheet.Class.
- Encapsulating the construction of the time record and establishing the association ensures that every paycheck has an associated time record.
- Code can be added to the Pays.Remove to destroy the corresponding objects, paycheck, and time_sheet, again to ensure that every paycheck has its required time sheet.

6.7 MANY CARDINALITY

Several options exist for implementing associations where more than one object is so associated with another object, e.g., in a many-to-one association. For the interface, a simple and effective approach is to use Ada's unconstrained array. For example, if the association Pays includes all of the paychecks approved for an employee, Pays is a many-to-one association:

```
with Check;
with Employee;
package Pays is
   type Check_List is array(Natural range <>) of Check.Class;

   procedure Establish (From : in [out] Check.Class;
                        To   : in       Employee.Class);
   procedure Establish (From : in [out] Check_List;
                        To   : in       Employee.Class);

   procedure Remove    (From : in     Check.Class;
                        To   : in     Employee.Class);
   procedure Remove    (From : in     Check_List;
                        To   : in     Employee.Class);

   function Pays        (From : in Check.Class)
      return Employee.Class;
   function Is_Payed_By (To   : in Employee.Class)
      return Check_List;

end Pays;
```

Here are some notes about this code:

- Both the single-valued and multi-valued forms of the Establish and Remove operations are provided. Providing both forms is optional. If the association is many-to-many, then the form with two arrays can be added.
- There is only one form of the traversal operations. In this case, returning only one of the checks would be an incomplete result, and traversing all of the associations of a list of checks may not result in a single employee.
- If the relation is conditional, the traversal operation that returns a list (Is_Payed_By) could return an empty list instead of raising an exception. However, to help avoid programming errors, the designer should choose a strategy for defining the behavior consistently for all operations.

The array approach is attractive for its simplicity. However, there are many alternatives and conventions for dealing with sets of values in Ada. The book *Data Structures Using Ada* [FELD85] presents several approaches and the corresponding design trade-offs.

6.8 OTHER ISSUES

Individual associations cannot be completely designed in isolation. For example, the associations for a particular set of classes may each be traversed in only one direction and a circularity can still exist. If there is a circularity, it must be broken by implementing some of the associations with an independent data structure; or as a last resort, all of the classes involved in the circular association can be defined in a single package.

The operations on relationships should have consistent semantics to avoid confusing the engineer that uses the operations. Behaviors that need definition include the following:

- What happens when a one-to-one association is established when one or both of the objects already have that association established?
- Does an attempt to remove an association that does not exist generate a constraint error?
- What exception is raised when an attempt is made to traverse a non-existent relationship?

Several details have been left out of previous discussions about implementing asso-

ciations. The behavior of the Destruct operation will depend on the cardinality of the association and the implementation. For example, if the association is a required association and the object is destroyed, tables may need to be updated or the related objects may need to be destroyed. The construction and destruction of objects will affect performance and may influence the choice of association implementation. The examples in this section assumed the use of Ada access types for the class type definitions. However, as discussed in other sections, alternatives can be used which may affect certain details of implementing associations.

Some approaches to object modeling allow associations to have attributes. The design and implementation issues for class attributes are very similar to those for association attributes. For example, whether an attribute should be defined in a separate package depends on the complexity of the attribute and how many classes (associations) might use the attribute. Of course, inheritance is not usually applied to associations, which simplifies the implementation of attributed associations considerably.

APPENDIX A.
OO PROGRAMMING EXAMPLE CODE (ADA 83)

A.1 ADA PACKAGE SPECIFICATIONS

A.1.1 ss_s.ada

```
--------------
-- Abstraction
--------------
package Social_Security is

  type Number is private;

  Default_Separator : constant Character := ' ';

  Invalid_Number : exception;
  function Construct (Part1 : in     Natural;
                      Part2 : in     Natural;
                      Part3 : in     Natural) return Number;

  function Image (Self      : in Number;
                  Separator : in Character := Default_Separator)
    return String;

private
  type Number is new String(1..11);
end Social_Security;
```

A.1.2 employee_s.ada

```
with Social_Security;
with ADAR_Comp;
--=======
-- Class:
--=======
package Employee is
  type Class is private;
  --============
  -- Attributes:
  --============
  type Name       is new String (1..25);
  type Number     is new Social_Security.Number;
  type Department is (Unknown,
```

```
                      Computer_Science,
                      Science_and_Technology,
                      Systems_Evaluation);
  type Status     is (Salaried, Hourly, Consultant);
  type Money      is new ADAR_Comp.Decimal (Precision => 9, Scale => 2);
  --==================
  -- Object Management:
  --==================
  -- Without a Constructor, this type cannot be used. Look at subclasses
  -- to see how to construct objects. In Ada 95, Class can be an
  -- abstract type!
  -- ============================
  -- Attribute access operations:
  -- ============================
  procedure Change (Self        : in Class;
                    Emp_Name    : in Name;
                    SS_Number   : in Number);
  -- Overloading is a form of (ad-hoc) polymorphism:
  procedure Change (Self : in Class; D      : in Department);
  procedure Change (Self : in Class; Salary : in Money);

  function Emp_Name       (Self : in Class) return Name;
  function SS_Number      (Self : in Class) return Number;
  function Emp_Status     (Self : in Class) return Status;
  function Emp_Department (Self : in Class) return Department;
  function Emp_Salary     (Self : in Class) return Money;

  --============
  -- Operations:
  --============
  package Abstract is
    function Net_Pay       (Self : in Class) return Money;
    function Send_Check_To (Self : in Class) return String;
  end Abstract;

----------------------------------------
-- For Child packages only (see Ada 95):
-- Start private section here when migrating to Ada 95!

  -- Status is used as the tag to simulate polymorphism.
  type Tag is new Status;
  type Structure (Tagged : Tag) is private;

private
  -- The discriminant, Tagged, is used to simulate runtime polymorphism.
  -- This should be replaced in Ada 9X with the corresponding tagged record
  -- declaration.
  type Structure (Tagged : Tag) is
    record
      Emp_Name       : Name;
      SS_Number      : Number;
      Emp_Department : Department;
      Emp_Salary     : Money;
```

```
    end record;

  type Class is access Structure;

end Employee;
```

A.1.3 employee_consulting_s.ada

```
with Employee;

--=========
-- Subclass
--=========
package Employee_Consulting is
  type Class is new Employee.Class; -- Inheritance

  --==================
  -- Object Management
  --==================
  procedure Initialize (Object : in out Class);

  --================================
  -- New attribute access operations
  --================================
  procedure Set_Mail_Address (Self : in Class; MA : in String);
  function  Mail_Address     (Self : in Class) return  String;

  --===============
  -- New operations
  --===============
  -- The Dispatching methods declared in the parent class must
  -- be defined for each subclass and the dispatching method itself
  -- updated to invoke the correct subclass method.
  function Net_Pay       (Self : in Class) return Employee.Money;
  function Send_Check_To (Self : in Class) return String;

-----------------------------------------------------------------------
-- For dispatching only. See implementation of Parent Class. Remove
-- in Ada 95.
  Unique_Tag : constant Employee.Tag := Employee.Consultant;

private
  -- A quick and dirty way to deal with unconstrained attributes:
  type Mailing_Address is access String;

  -- The subclass structure must keep the parent structure intact
  -- while appending additional data. This implementation works for
  -- most compilers:
  type Structure is
    record
      Parent : Employee.Structure(Unique_Tag);
      MA     : Mailing_Address;
    end record;
```

```
end Employee_Consulting;
```

A.1.4 employee_taxable_s.ada

```
with Employee;

--=========
-- Subclass
--=========
-- This package combines two subclasses, something Ada can do more
-- conveniently than other languages:
package Employee_Taxable is
  type Class is new Employee.Class;

  --============
  -- Attributes:
  --============
  -- Tax uses the already defined Employee.Money type.
  type Deduction is range 0..12;

  --===================
  -- Object management:
  --===================
  procedure Initialize_Hourly   (Object : in out Class);
  procedure Initialize_Salaried (Object : in out Class);

  --==========================
  -- New attribute operations:
  --==========================
  procedure Change (Self : in Class; Tax : in Employee.Money);
  procedure Change (Self : in Class; D   : in Deduction);

  function Tax        (Self : in Class) return Employee.Money;
  function Deductions (Self : in Class) return Deduction;

  --================
  -- New operations:
  --================
  function Net_Pay       (Self : in Class) return Employee.Money;
  function Send_Check_To (Self : in Class) return String;

------------------------------------------------------------
  Unique_Hourly_Tag   : constant Employee.Tag := Employee.Hourly;
  Unique_Salaried_Tag : constant Employee.Tag := Employee.Salaried;
private
  -- The two different structures are identical except for the tag:
  type Structure_Hourly is   -- new Employee.Structure with
    record
      Parent : Employee.Structure(Unique_Hourly_Tag);
      Tax    : Employee.Money;
      D      : Deduction;
    end record;
```

```
  type Structure_Salaried is -- new Employee.Structure with
    record
      Parent : Employee.Structure(Unique_Salaried_Tag);
      Tax    : Employee.Money;
      D      : Deduction;
    end record;
end Employee_Taxable;
```

A.1.5 employee_file_s.ada

```
with Employee; -- An associated class

--======
-- Class
--======
package Employee_File is
  type Class is limited private;

  --====================
  -- Object management:
  --====================
  -- The Open procedure provides the constructor method:
  Unable_to_Open_File : exception;
  procedure Open  (Self      : in out Class;
                   Path_Name : in     String);

  -- The Close procedure provides the destructor method:
  procedure Close (Self : in out Class);

  --============
  -- Operations:
  --============
  Unable_to_Read_File      : exception;
  Attempt_to_Read_Past_EOF : exception;
  function Get_Next (Self   : in Class) return Employee.Class;

  function End_of_File (Self : in Class) return Boolean;

private
  -- When Structure is not visible, cannot inherit from this class. A tag
  -- is not needed, either.
  type Structure;

  type Class is access Structure;
end Employee_File;
```

A.1.6 check_s.ada

```
with Employee; -- an associated class
with Calendar; -- used for the date attribute
```

```
--======
-- Class
--======
package Check is
  type Class is private;

  --====================
  -- Object management:
  --====================
  -- This class has an association with Employee.Class which is
  -- implemented one way.
  function Construct (Pays   : in Employee.Class;
                      Number : in Natural;
                      Date   : in Calendar.Time := Calendar.Clock)
    return Class;

  --===========
  -- Operations
  --===========
  procedure Print (Self : in Class);

private
  type Structure is
    record
      Pays   : Employee.Class;
      Number : Natural;
      Date   : Calendar.Time;
    end record;

  type Class is access Structure;

end Check;
```

A.1.7 payroll.ada

```
with Employee_File;
with Check;
with Tax_Calculation;

with Text_IO; -- For User Interface

--======
-- Class
--======
-- This is a control class, most easily implemented as a procedure,
-- although a package could be used in preparation for a more
-- sophisticated user interface (such as X-windows callbacks).
procedure Payroll is
  -- There is only one payroll, therefore a type definition is not needed.

  --===========
  -- Attributes
```

```
  --============
  DB           : Employee_File.Class;
  Check_Number : Natural;
  Report_File_Name : constant String := "PRINTER.DAT";

  -- User-Interface
  Input_Buffer : String (1..80);
  Input_Length : Natural;

  --============
  -- Operations
  --============
begin
  Generate_Report:
    begin
      Tax_Calculation;
    end Generate_Report;

  Set_File_Name:
    begin
      Employee_File.Open (DB, Report_File_Name);
    end Set_File_Name;

  Set_Starting_Check_Number:
    begin
      Text_IO.Put_Line ("Enter starting check number:");
      Text_IO.Get_Line (Input_Buffer, Input_Length);
      Check_Number := Natural'Value (Input_Buffer (1..Input_Length));
    end Set_Starting_Check_Number;

  Print_Payroll:
    begin
      while not Employee_File.End_of_File (DB) loop
        begin
          Check.Print (Check.Construct (Employee_File.Get_Next (DB),
                                        Check_Number));
          Check_Number := Check_Number + 1;
        exception
          when Employee_File.Attempt_to_Read_Past_EOF =>
            exit;
          when others =>
            null;
        end;
      end loop;

      Employee_File.Close (DB);
    end Print_Payroll;

end Payroll;
```

A.2 ADA PACKAGE BODIES

A.2.1 ss_b.ada

```
package body Social_Security is
  ---------------------
  Operation definitions
  ---------------------
  function Fixed_Image (N      : in Natural;
                        Length : in Natural) return String is
    Result : String(1..Length) := String'(1..Length => '0');
    Image  : constant String := Natural'Image(N);
    L      : constant Natural := Image'Length;
  begin
    Result(2+Length-L..Length) := Image(Image'First+1..Image'Last);
    return Result;
  end Fixed_Image;
  ----------
  function Construct (Part1 : in     Natural;
                      Part2 : in     Natural;
                      Part3 : in     Natural) return Number is
  begin
    if Part1 in 0..999  and
       Part2 in 0..99   and
       Part3 in 0..9999 then
      return Number (Fixed_Image (Part1,3) & '-' &
                     Fixed_Image (Part2,2) & '-' &
                     Fixed_Image (Part3,4)
                    );
    else
      raise Invalid_Number;
    end if;
  end Construct;
  ----------
  function Image (Self      : in Number;
                  Separator : in Character := Default_Separator)
    return String is
  begin
    if Separator = Default_Separator then
      return String(Self);
    else

      Convert_Delimeter:
      declare
        Str : String(1..Self'Length) := String(Self);
      begin
        Str(4) := Separator;
        Str(7) := Separator;
        return Str;
      end Convert_Delimeter;

    end if;
  end Image;
```

```
end Social_Security;
```

A.2.2 employee_b.ada

```
package body Employee is
  ---------------------
  Operation definitions
  ---------------------
  procedure Change (Self        : in Class;
                    Emp_Name    : in Name;
                    SS_Number   : in Number) is
  begin
    Self.Emp_Name  := Emp_Name;
    Self.SS_Number := SS_Number;
  end;
  ----------
  procedure Change (Self : in     Class; D : in     Department) is
  begin
    Self.Emp_Department := D;
  end;
  ----------
  procedure Change (Self : in     Class; Salary : in     Money) is
  begin
    Self.Emp_Salary := Salary;
  end;
  ----------
  function Emp_Name  (Self : in Class) return Name is
  begin
    return Self.Emp_Name;
  end;
  ----------
  function SS_Number (Self : in Class) return Number is
  begin
    return Self.SS_Number;
  end;
  ----------
  function Emp_Status     (Self : in Class) return Status is
  begin
    return Status(Self.Tagged);
  end;
  ----------
  function Emp_Department (Self : in Class) return Department is
  begin
    return Self.Emp_Department;
  end;
  ----------
  function Emp_Salary     (Self : in Class) return Money is
  begin
    return Self.Emp_Salary;
  end;
  -- Dispatching operations can be separate to make updates easier
  -- and to localize the context clauses to the operations that use them.
```

```
  package body Abstract is separate;

end Employee;
```

A.2.3 employee_consulting_b.ada

```
with Unchecked_Conversion; -- For simulating inheritance
with Ada; -- .Tags

package body Employee_Consulting is
  --=======================
  -- Subclass Implementation
  --=======================
  Data type definition
  --------------------
  type Child_Pointer is access Structure; -- of the Child.
  --------------------
  Operation definitions
  ---------------------
  function Narrow (Parent_Pointer : in Class) return Child_Pointer is

    function Convert_Pointer is
      new Unchecked_Conversion (Source => Class,
                                Target => Child_Pointer);

    Result : constant Child_Pointer
           := Convert_Pointer (Parent_Pointer);
    use Employee;
  begin
    if Result.Parent.Tagged = Unique_Tag then -- of this Child
      return Result;
    else
      raise Ada.Tags.Tag_Error;
    end if;
  end Narrow;
  ----------
  procedure Initialize (Object : in out Class) is
    function Convert_Pointer is
      new Unchecked_Conversion (Child_Pointer, Class);
  begin
    Object := Convert_Pointer (new Structure); -- of the Child
  end;
  ----------
  function Mail_Address (Self : in Class) return String is
    P : constant Child_Pointer := Narrow(Self);
  begin
    if P.MA = null then  return “Hold”;
    else                 return P.MA.all;
    end if;
  end;
  ----------
  procedure Set_Mail_Address (Self : in     Class; MA : in String) is
  begin
```

```
    Narrow(Self).MA := new String'(MA);
  end;
  ----------
  function Net_Pay       (Self : in Class) return Employee.Money is
  begin
    return Emp_Salary(Self);
  end;
  ----------
  function Send_Check_To (Self : in Class) return String is
  begin
    return Mail_Address(Self);
  end;

end Employee_Consulting;
```

A.2.4 employee_taxable_b.ada

```
with Unchecked_Conversion; -- For simulating inheritance
with Ada; --.Tags

package body Employee_Taxable is
  --========================
  -- Subclass Implementation
  --========================
  Data type definitions
  ---------------------
  type Pointer_H is access Structure_Hourly;
  type Pointer_S is access Structure_Salaried;
  --------------------
  Operation definitions
  ---------------------
  -- Narrow arbitrarily uses Pointer_H, it could use Pointer_S:
  function Narrow (Parent_Pointer : in Class) return Pointer_H is
    function Convert_Pointer is
      new Unchecked_Conversion (Source => Class,
                                Target => Pointer_H);
    Result : constant Pointer_H := Convert_Pointer(Parent_Pointer);
    use Employee; -- for "=" operator
  begin
    if Result.Parent.Tagged = Hourly or
       Result.Parent.Tagged = Salaried then
      return Result;
    else
      raise Ada.Tags.Tag_Error;
    end if;
  end Narrow;
 ----------
  procedure Initialize_Hourly (Object : in out Class) is

    function Convert_Pointer is
      new Unchecked_Conversion (Pointer_H, Class);
```

```
  begin
    Object := Convert_Pointer (new Structure_Hourly);
  end;
 -----------
  procedure Initialize_Salaried (Object : in out Class) is

    function Convert_Pointer is
      new Unchecked_Conversion (Pointer_S, Class);
  begin
    Object := Convert_Pointer (new Structure_Salaried);
  end;
 -----------
  procedure Change (Self : in Class; Tax : in     Employee.Money) is
  begin
    Narrow(Self).Tax := Tax;
  end;
 -----------
  procedure Change (Self : in Class; D   : in Deduction) is
  begin
    Narrow(Self).D := D;
  end;
 -----------
  function Tax        (Self : in Class) return Employee.Money is
  begin
    return Narrow(Self).Tax;
  end Tax;
 -----------
  function Deductions (Self : in Class) return Deduction is
  begin
    return Narrow(Self).D;
  end Deductions;
 -----------
  function Net_Pay       (Self : in Class) return Employee.Money is
    Result : Employee.Money := Emp_Salary(Self);
    use Employee;
  begin
    Decrement (Result, Tax(Self), Rounded => True);
    return Result;
  end;
 -----------
  function Send_Check_To (Self : in Class) return String is
  begin
    return Employee.Department'Image (Emp_Department(Self));
  end;

end Employee_Taxable;
```

A.2.5 empolyee_abstract.ada

```
with Employee_Taxable;
with Employee_Consulting;

separate (Employee)
```

```
package body Abstract is
 ----------------------
  Operation definitions
 ----------------------
  function Net_Pay (Self : in Class) return Money is
  begin -- Dispatching
    case Self.Tagged is
      when Hourly | Salaried =>
        return Employee_Taxable.Net_Pay
          (Employee_Taxable.Class   (Self));

      when Consultant        =>
        return Employee_Consulting.Net_Pay
          (Employee_Consulting.Class(Self));
      -- Add additional children here
      -- No others clause! This is an abstract operation!
   end case;
  end;
 -----------
  function Send_Check_To (Self : in Class) return String is
  begin
    case Self.Tagged is
      when Hourly | Salaried =>
        return Employee_Taxable.Send_Check_To
                (Employee_Taxable.Class   (Self));

      when Consultant        =>
        return Employee_Consulting.Send_Check_To
                (Employee_Consulting.Class(Self));

      -- Add additional children here
      -- No others clause! This is an abstract operation!

    end case;
  end;

end Abstract;
```

A.2.6 employee_file_b.ada

```
with Employee_Taxable;
with Employee_Consulting;

with ADAR_Comp;
with Sequential_IO;

-- Most of this file involves parsing an ASCII text file.
package body Employee_File is

-- COBOL specification of data file format:
--01      DETAIL-LINE.
--        02      DL-NAME                 PIC     X(25).
--        02      FILLER                  PIC     X(8)    VALUE   SPACES.
```

```
--        02      DL-EMPLOYEE-NUMBER     PIC     X(9).
--        02      FILLER                 PIC     X(9)    VALUE   SPACES.
--        02      DL-STATUS              PIC     X.
--        02      FILLER                 PIC     X(7)    VALUE   SPACES.
--        02      FILLER                 PIC     X(6)    VALUE   SPACES.
--        02      DL-DEPARTMENT          PIC     X(2).
--        02      FILLER                 PIC     X(5)    VALUE   SPACES.
--        02      DL-DEPENDENT           PIC     Z9.
--        02      FILLER                 PIC     X(12)   VALUE   SPACES.
--        02      DL-SALARY              PIC     $ZZZ,ZZZ.99.
--        02      FILLER                 PIC     X(8)    VALUE   SPACES.
--        02      DL-TAX                 PIC     $ZZZ,ZZZ.99.
 ----------------------
 Data type definitions
 ----------------------
  type Detail_Line is
    record
      Emp_Name       : String (1..25);
      Filler_1       : String (1..8);
      Emp_Number     : String (1..9);
      Filler_2       : String (1..9);
      Emp_Status     : Character;
      Filler_3       : String (1..13);
      Emp_Department : String (1..2);
      Filler_4       : String (1..5);
      Emp_Deductions : String (1..2);
      Filler_5       : String (1..12);
      Emp_Salary     : String (1..11);
      Filler_6       : String (1..8);
      Tax            : String (1..11);
      Filler_7       : String (1..16);
    end record;

  type Status_Conversion is array (Character) of Employee.Status;
  Convert_Status : constant Status_Conversion
      := Status_Conversion'('s' | 'S' |'f' | 'F' => Employee.Salaried,
                                      'h' | 'H' => Employee.Hourly,
                                      'c' | 'C' => Employee.Consultant,
                                      others    => Employee.Hourly);

  type Department_Code is (CS,ST,SE);
  type Conversion is array (Department_Code) of Employee.Department;
  Department_Convert : constant Conversion
                     := (CS => Employee.Computer_Science,
                         ST => Employee.Science_and_Technology,
                         SE => Employee.Systems_Evaluation);

  package File_Operations is
    new Sequential_IO (Detail_Line);

  type Structure is
    record
      File : File_Operations.File_Type;
```

```
  end record;
---------------------
Operation definitions
---------------------
procedure Open  (Self      : in out Class;
                 Path_Name : in     String) is
begin
  if    Self = null        then Self := new Structure;
  elsif File_Operations.Is_Open (Self.File) then
    File_Operations.Close (Self.File);
  end if;

  File_Operations.Open (File => Self.File,
                        Mode => File_Operations.In_File,
                        Name => Path_Name);
exception
  when others =>
    raise Unable_to_Open_File;
end Open;
----------
procedure Close (Self : in out Class) is
begin
  if Self /= null then [the slash?]
      File_Operations.Close (Self.File);
      -- Deallocate Self -- TBD
  end if;
end Close;
----------
function Salary_Value (S : in String) return Employee.Money is
  Parse_S : String(1..S'Length) := S;
  Result  : Employee.Money;
begin
  Zero_Leading:
    for I in Parse_S'Range loop
      case Parse_S(I) is
        when '$' | '*' | ' ' => Parse_S(I)    := '0';
        when ','             => Parse_S(2..I) := Parse_S(1..I-1);
                                Parse_S(1)    := ' ';
        when others          => null;
      end case;
    end loop Zero_Leading;

  Employee.Move (Parse_S, Result);
  return Result;
end Salary_Value;
----------
function Construct (Tag  : in Employee.Status;
                    Name : in Employee.Name;
                    SS   : in Employee.Number)
  return Employee.Class is
  Result : Employee.Class;
  use Employee;
begin
```

```
  case Tag is
    when Consultant => Employee_Consulting.Initialize
      (Employee_Consulting.Class(Result));
    when Salaried   => Employee_Taxable.Initialize_Hourly
      (Employee_Taxable.Class(Result));
    when Hourly     => Employee_Taxable.Initialize_Salaried
      (Employee_Taxable.Class(Result));
  end case;
  Employee.Change (Result, Name, SS);

  return Result;

end;
----------
function Get_Next  (Self   : in Class) return Employee.Class is
  Data_Record : Detail_Line;
begin
  if File_Operations.End_of_File (Self.File) then
    raise Attempt_to_Read_Past_EOF;
  end if;

  File_Operations.Read (Self.File, Data_Record);

  Parse_Data_Record:
  declare
    Result : constant Employee.Class := Construct (
               Tag  => Convert_Status(Data_Record.Emp_Status),
               Name => Employee.Name(Data_Record.Emp_Name),
               SS   => Employee.Construct (
                Part1 => Natural'Value (Data_Record.Emp_Number(1..3)),
                Part2 => Natural'Value (Data_Record.Emp_Number(4..5)),
              Part3 => Natural'Value (Data_Record.Emp_Number(6..9))));
  begin
    Employee.Change (Result, Salary_Value(Data_Record.Emp_Salary));

    Determine_Department:
    declare
      D : Department_Code;
    begin
      D := Department_Code'Value (Data_Record.Emp_Department);
      Employee.Change (Result, Department_Convert (D));
    exception
      when others => Employee.Change (Result, Employee.Unknown);
    end Determine_Department;

    case Employee.Emp_Status(Result) is
      when Employee.Salaried | Employee.Hourly =>
       Employee_Taxable.Change (Self => Employee_Taxable.Class(Result),
                                Tax  => Salary_Value (Data_Record.Tax));
        Employee_Taxable.Change (Employee_Taxable.Class(Result),
        Employee_Taxable.Deduction'Value(Data_Record.Emp_Deductions));
      when others =>
        null;
```

```
      end case;

      return Result;
    end Parse_Data_Record;

  exception
    when others =>
      raise Unable_to_Read_File;
  end Get_Next;
  ----------
  function End_of_File (Self : in Class) return Boolean is
  begin
    return Self = null or else
           not File_Operations.Is_Open (Self.File) or else
           File_Operations.End_Of_File (Self.File);
  end End_of_File;

end Employee_File;
```

A.2.7 check_b.ada

```
with Text_IO;   -- To print the check
with ADAR_Comp; -- Format money

package body Check is
  ---------------------
  Operation definitions
  ---------------------
  function Construct (Pays   : in Employee.Class;
                      Number : in Natural;
                      Date   : in Calendar.Time := Calendar.Clock)
    return Class is
    Result : Class := new Structure;
  begin
    Result.Pays   := Pays;
    Result.Number := Number;
    Result.Date   := Date;

    return Result;
  end Construct;
  ----------
  function Date_Image (Date : Calendar.Time) return String is
    use Calendar;
  begin
    return Month_Number'Image(Month(Date)) & '/' &
           Day_Number'Image  (Day  (Date)) & '/' &
           Year_Number'Image (Year (Date));
  end;
  ----------
  procedure Print (Self : in Class) is
  begin
    Text_IO.New_Line;
```

```
    Text_IO.Put_Line(Integer'Image(Self.Number) &
       String'(1..10 => ' ') & Date_Image(Self.Date));
    Text_IO.New_Line;

    Text_IO.Put_Line (String(Employee.Emp_Name(Self.Pays)) &
         String'(1..5 => ' ') &
        '$' & Employee.Image(Employee.Abstract.Net_Pay(Self.Pays)));

    Text_IO.New_Line;
    Text_IO.Put_Line ("Send Check to:" &
                      Employee.Abstract.Send_Check_To(Self.Pays));
    Text_IO.New_Line;
  end;

end Check;
```

LIST OF REFERENCES

[ANSI95] ANSI/ISO/IEC-8652:1995, *Ada 95 Reference Manual: The Language*, The Standard Libraries, American National Standards Institute, New York, NY, January 1995.

[BAKR91] H. G. Baker, "Object-Oriented Programming in Ada83—Genericity Rehabilitated," *ACM Ada Letters*, Vol. XI, No. 9, 1991, pp. 116-127.

[BARN93a] J. Barnes, *Introducing Ada 9X: Ada 9X Project Report*, prepared for the Office of the Under Secretary of Defense for Acquisition by Intermetrics, Inc., Intermetrics, Washington, DC, 1993.

[BARN93b] J. Barnes, "Introducing Ada 9X," *ACM Ada Letters*, Vol. XIII, No. 6, November/December 1993.

[BOO87] G. Booch, *Software Components with Ada*, Benjamin/Cummings, Redwood City, CA, 1987.

[BOO94] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, CA, 1994.

[DOD93a] Department of Defense, DoD Directive 8120.1, *Life-Cycle Management (LCM) of Automated Information Systems (AISs)*, January 14, 1993.

[DOD93b] Department of Defense, DoD Instruction 8120.2, *Automated Information System Life-Cycle Management Process, Review, and Milestone Approval Procedures*, January 14, 1993.

[CAWE85] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys* Vo.l 17, No. 4, December 1985, p. 481. Quoted in G. Booch, *Software Components with Ada*, Benjamin/ Cummings, Redwood City, CA, 1987.

[CDYD91] P. Coad and E. Yourdon, *Object-Oriented Analysis*, 2nd edition, Yourdon Press, Englewood Cliffs, NJ, 1991.

[FELD85] M. Feldman, *Data Structures with Ada*, Addition-Wesley, Reading, MA, 1985.

[HIRA92] M. Hirasuna, "Using Inheritance and Polymorphism with Ada in Government Sponsored Contracts," *ACM Ada Letters*, XII, 2, 1992, pp. 43-56.

[HIRA94a] M. Hirasuna, "An Ada 9X Subset for Inheritance-Based Reuse and its Translation to Ada 83 (Part 1)," *ACM Ada Letters*, Vol. XIV, No. 1, 1994, pp. 50-60.

[HIRA94b] M. Hirasuna, "An Ada 9X Subset for Inheritance-Based Reuse and Its Translation to Ada 83 (Part 2)," *ACM Ada Letters*, Vol. XIV, No. 2, 1994, pp. 58-67.

[HUTT94] A. T. F. Hutt, editor, *Object Analysis and Design: Description of Methods*, John Wiley & Sons, New York, NY, 1994.

[IDA95a] B. A. Haugh, M. C. Frame, and K. Jordan, *An Object-Oriented Development Process for Department of Defense Information Systems*, IDA Paper P-3142, Institute for Defense Analyses, Alexandria, VA, July 1995.

[IDA95b] D. Smith, B. Haugh, K. Jordan, *Object-Oriented Programming Strategies for Ada*, IDA Paper P-3143, Institute for Defense Analyses, Alexandria, VA, July 1995.

[IDA95c] B. Haugh, A. Noor, D. Smith, K. Jordan, *Legacy System Wrapping for DoD Information System Modernization*, IDA Paper P-3144, Institute for Defense Analyses, Alexandria, VA, July 1995.

[IDA95d] K. Jordan, B. Haugh, *Software Reengineering Using Object-Oriented Technology*, IDA Paper P-3145, Institute for Defense Analyses, Alexandria, VA, July 1995.

[JHNS93] H. Johansson, "Object Oriented Programming and Virtual Functions in Conventional Languages; an Extended Abstract," *ACM Ada Letters*, Vol. XIII, No. 4, 1993, pp. 44-48.

[RUMB91] J. Rumbaugh, M. Blaha, W. Premerlani, E. Frederick, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[SEID92] E. Seidewitz, “Object-Oriented Programming with Mixins in Ada,” *ACM Ada Letters*, Vol. XII, No. 2, 1992, pp. 76-90.

[WEG90] P. Wegner, “Concepts and Paradigms of Object-Oriented Programming,” *OOPS Messenger* Vol. 1/No. 1, August 1990.

GLOSSARY

Words used in the definition of a glossary term and that are defined elsewhere are in **bold**.

Abstraction — Abstraction consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental properties [RUMB91].

AIS Program — A directed and funded AIS effort, to include all **migration** systems, that is designed to provide a new or improved capability in response to a validated need [DOD93a].

Architecture — The organizational structure of a system or **CSCI**, identifying its components, their interfaces, and a concept of execution among them [DOD94a].

Automated Information System (AIS) — A combination of **computer hardware** and computer **software**, data, and/or telecommunications that performs functions such as collecting, processing, transmitting, and displaying information. Excluded are computer resources, both hardware and software, that are either physically part of, dedicated to, or essential in real time to the mission performance of weapon systems; used for **weapon system** specialized training, simulation, diagnostic test and maintenance, or calibration; or used for research and development of weapon systems [DOD93a]. However, as used here, AISs include systems for C2I, C3I, and C4I, even though they may be essential in real time to mission performance.

Class — A class can be defined as a description of similar **object**s, like a template or cookie cutter [NEL91]. The class of an object is the definition or description of those attributes and behaviors of interest.

Collaboration — A request from a client to a server in fulfillment of a client's responsibilities [HUTT94, p. 192].

Commercial-off-the-Shelf (COTS) — Commercial items that require no unique government modifications or maintenance over the life cycle of the product to meet the needs of the procuring agency [DOD93a].

Computer Hardware — Devices capable of accepting and storing computer data, executing a systematic sequence of operations and computer data, or producing control outputs. Such devices can perform substantial interpretation, computation, communication, control, or other logical functions [DOD94a].

Computer Program — A combination of computer instructions and data definitions that enables **computer hardware** to perform computational or control functions [DOD94a].

Computer Software Configuration Item (CSCI) — An aggregation of software that satisfies an end use function and is designated for separate configuration management by the acquirer. CSCIs are selected based on tradeoffs among software function, size, host or target computers, developer, support concept, plans for reuse, criticality, interface considerations, [the] need to be separately documented and controlled, and other factors [DOD94a].

Contract — The list of requests that a client **class** can make of a server class. Both must fulfill the contract: the client by making only those requests the contract specifies, and the server by responding appropriately to those requests [HUTT94, p. 192].

CRC Cards — **Class-Responsibility**-Collaborator Cards. CRC cards are pieces of paper divided into three areas: the **class** name and the purpose of the class, the responsibilities of the class, and the collaborators of the class. CRC cards are intended to be used to iteratively simulate different scenarios of using the system to get a better understanding of its nature [HUTT94, p. 192].

Database — A collection of related data stored in one or more computerized files in a manner that can be accessed by users or computer programs via a database management system [DOD94a].

Database Management System — An integrated set of computer programs that provide the capabilities needed to establish, modify, make available, and maintain the integrity of a database [DOD94a].

Encapsulation — . . . (also **information hiding**) consists of separating the external aspects of an **object**, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects [RUMB91]. The act of grouping into a single object both data and the operation that affects that data [WIR90].

Framework — A collection of **class** libraries, generics, design, scenario models, documentation, etc., that serves as a platform to build applications.

Government-off-the-Shelf (GOTS) — Products for which the Government owns the data rights, that are authorized to be transferred to other DoD or Government customers, and that require no unique modifications or maintenance over the life cycle of the product [DOD93b].

Inheritance — Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship [RUMB91]. Subclasses of a **class** inherit the operations of the parent class and may add new operations and new instance variables. Inheritance allows us to reuse the behavior of a class in the definition of new classes [WEG90].

Information Hiding — Making the internal data and methods inaccessible by separating the external aspects of an **object** from the internal (hidden) implementation details of the object.

Information System — See **Automated Information System (AIS)**.

Legacy System — Any currently operating automated system that incorporates obsolete computer technology, such as proprietary hardware, closed systems, “stovepipe” design, or obsolete programming languages or **database** systems.

Life-Cycle Management (LCM) — A management process, applied throughout the life of an **AIS**, that bases all programmatic decisions on the anticipated mis-

	sion-related and economic benefits derived over the life of the AIS [DOD93a].
Message	Mechanism by which **object**s in an OO system request **service**s of each other. Sometimes this is used as a synonym for **operation**.
Method	An operation upon an **object**, defined as part of the declaration of a **class**; all methods are **operation**s, but not all operations are methods [BOO94a].
Migration	The transition of support and operations of software functionality from a **legacy system** to a **migration system**.
Migration System	An existing **AIS**, or a planned and approved AIS, that has been officially designated to support standard processes for a functional activity applicable DoD-wide or DoD Component-wide [DOD93a]. Ordinarily, an AIS that has been designated to assume the functionality of a legacy AIS.
Monomorphism	A concept in type theory, according to which a name (such as a variable declaration) may only denote **object**s of the same **class** [BOO94a].
Object	A combination of state and a set of methods that explicitly embodies an **abstraction** characterized by the behavior of relevant requests. An **object** is an instance of an implementation and an interface. An object models a real-world entity (such as a person, place, thing, or concept), and it is implemented as a computational entity that encapsulates state and operations (internally implemented as data and methods) and responds to requestor services.
Object-Based Programming	A method of programming in which programs are organized as cooperative collections of objects, each of which represents an instance of some type, and whose types are all members of a hierarchy of types . . . somewhat constrained by the existence of static binding and **monomorphism** [BOO94a].

.

Object-Oriented Analysis	A method of analysis in which requirements are examined from the perspective of the classes and objects found in the vocabulary of the problem domain [BOO94a].
Object-Oriented Decomposition	The process of breaking a system into parts, each of which represents some **class** or **object** from the problem domain [BOO94a].
Object-Oriented Design	A method of design encompassing the process of OO decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design [BOO94a].
Object-Oriented Programming	A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some **class**, and whose classes are members of a hierarchy of classes united via **inheritance** relationships. In such programs, classes are generally viewed as static, whereas objects typically have a much more dynamic nature, which is encouraged by the existence of dynamic binding and **polymorphism** [BOO94a].
Object-Oriented Technology (OOT)	OOT consists of a set of methodologies and tools for developing and maintaining software systems using software objects composed of encapsulated data and operations as the central paradigm.
Object Request Broker (ORB)	Program that provides a location and implementation-independent mechanism for passing a **message** from one **object** to another.
Operation	Some action that one object performs upon another in order to elicit a reaction [BOO91]. A Service is a specific behavior that an Object is responsible for exhibiting [CDYD91].
Polymorphism	The same operation may behave differently on different classes [RUMB91].
Reengineering	The process of examining and altering an existing system to reconstitute it in a new form. May include reverse engineering

(analyzing a system and producing a representation at a higher level of **abstraction**, such as design from code), restructuring (transforming a system from one representation to another at the same level of abstraction), redocumentation (analyzing a system and producing user or support documentation), forward engineering (using software products derived from an existing system, together with new requirements, to produce a new system), retargeting (transforming a system to install it on a different target system), and translation (transforming source code from one language to another or from one version of a language to another) [DOD94a].

Requirement (1) Characteristic that a system or **CSCI** must possess in order to be acceptable to the acquirer. (2) A mandatory statement in this standard or another portion of the **contract** [DOD94a].

Responsibility A **contract** that a **class** must support, intended to convey a sense of the purpose of the class and its place in the system [HUTT94, p. 192].

Service A service is a specific behavior that an Object is responsible for exhibiting [CDYD91].

Software Computer programs and computer databases. **Note:** Although some definitions of software includes documentation, MIL-STD-498 limits the scope of this term to computer programs and computer databases in accordance with Defense Federal Acquisition Regulation Supplement 227.401 [DOD94a].

Software Development A set of activities that results in **software** products. Software development may include new development, modification, reuse, **reengineering**, or any other activities that result in software products [DOD94a].

Software Engineering In general usage, a synonym for **software development**. As used in this standard [MIL-STD 498], a subset of software development consisting of all activities except qualification testing. The standard makes this distinction for the sole purpose of giving separate names to the software engineering and software

	test environments [DOD94a].
Software Engineering Environment	The facilities, hardware, software, firmware, procedures, and documentation needed to perform **software engineering**. Elements may include but are not limited to computer-aided software engineering (CASE) tools, compilers, assemblers, linkers, loaders, operating systems, debuggers, simulators, emulators, documentation tools, and **database** management systems [DOD94a].
Software System	A system consisting solely of software and possibly the computer equipment on which the software operates [DOD94a].
Weapon System	Items that can be used directly by the Armed Forces to carry out combat missions and that cost more than 100,000 dollars or for which the eventual total procurement cost is more than 10 million dollars. That term does not include commercial items sold in substantial quantities to the general public (Section 2403 of 10 U.S.C., reference (bb) [DOD93a].

LIST OF ACRONYMS

ADAR	Ada Decimal Arithmetic and Representatives
C2	Command and Control
C3	Command, Control, and Communications
C4I	Command, Control, Communications, Computers, and Intelligence
CASE	Computer-Aided Software Engineering
DoD	Department of Defense
HLU	Hierarchy Library Units
IDA	Institute for Defense Analyses
OO	Object Oriented
OODBMS	Object-Oriented Database Management System
OOP	Object-Oriented Programming
OOT	Object-Oriented Technology